2005:157 CIV

EXAMENSARBETE

A Study of Sensory-based Control and Mechatronic Systems for Hybrid Rocket Operation

CONNY GUSTAFSSON

MASTER OF SCIENCE PROGRAMME
in Space Engineering

Luleå University of Technology
Department of Space Science, Kiruna

LULEÅ TEKNISKA UNIVERSITET

2005:157 CIV • ISSN: 1402 - 1617 • ISRN: LTU - EX - - 05/157 - - SE

A Study of Sensory-based Control and Mechatronic Systems for Hybrid Rocket Operation

Conny Gustafsson

Abstract

This Master's Thesis concerns the control of a small-scale autonomous hybrid rocket. It is an investigation of how an onboard computer system can be used for performing the low-level activities required for controlling the rocket in flight, e.g. data acquisition, inertial navigation, propulsion and attitude control.

The Thesis contains an in-depth study of the underlying physics of rocket motion as well as numerical simulations using MATLAB Simulink. The simulations indicate that even complex systems such as rocket attitude control can be performed using simple control principles.

The thesis also explains how to implement the required software elements using the QNX real-time operating system and the C programming language.

MATLAB® is a registered trademark of The Mathoworks, Inc.
QNX® is a registered trademark of QNX Software Systems, Ltd.

Preface and acknowledgements

The research leading to this thesis has been performed at the Robotics and Mechatronics Research Laboratory (RMRL) at Monash University, Melbourne, Australia, from March to August 2003.

I would like to thank in particular my supervisor at Monash University, Associate Professor Bijan Shirinzadeh for always finding the time, and for all the help and insight he provided.

I also thank Dr. Damon Honnery for his help with the hybrid rocket motor and for letting me in to the combustions lab as a visitor at the rocket launches, Professor Julio Soria, who helped me during those long hours with the Linux operating system and provided me with information on parallel port and real-time programming and my supervisor at Luleå, Dr. Björn Graneli for his long patience, and for putting confidence in me.

Special thanks to Sven Molin, coordinator of the Luleå - Monash exchange program who helped me with the preparations before my travel and who offered me a place to stay while in Melbourne.

Also, thanks to everyone at RMRL for their help and for the after-hours of socializing at the local pub. And thanks to Mark Symmonds for his help with the electronic components.

Finally, I especially want to thank my friends and my family for all their help and support.

Conny Gustafsson

List of Acronyms

A/D – Analog to digital

ADC – Analog to Digital Converter

API – Application Programming Interface

AT – Advanced Technology

COM – Common Object Model

CPU – Central Processing Unit

DAQ – Data Acquisition

DHCP – Dynamic Host Configuration Protocol

DMA – Direct Memory Access

FET – Field-Effect Transistor

FFT – Fast Fourier Transform

FTP – File-transfer Protocol

GUI – Graphical User Interface

I/O – Input/Output

IDE – Integrated Device Electronics

IMU – Inertial Measurement Unit

IRQ – Interrupt Request

ISA – Industry Standard Architecture

LAN – Local Area Network

OS – Operating System

PC – Personal Computer

PCI – Peripheral Component Interconnect

PID – Proportional-Integral-Derivative

PWM – Pulse-Width Modulation

RAM – Random Access Memory

R/W-Lock – Read/Write-Lock

SBC – Single Board Computer

SCP – Secure Copy Protocol

SDRAM – Synchronous Dynamic Random Access Memory

SSH – Secure Shell

TCP/IP – Transmission Control Protocol/Internet Protocol

TTL – Transistor-Transistor Logic

WLAN – Wireless Local Area Netwo

Table of Contents

1 Introduction 1
1.1 Rocket modules 1
1.1.1 Rocket casing 1
1.1.2 Hybrid rocket motor 2
1.1.3 Inertial measurement unit 2
1.1.4 Attitude and guidance system 2
1.1.5 Power supply 2
1.1.6 Parachute 3
1.2 Problem summary 3
1.2.1 Propulsion control 3
1.2.2 Data acquisition 3
1.2.3 Attitude control 3
1.2.4 Data storage and communication 3
1.3 Report layout 4
2 Setup 5
2.1 What is PC/104? 5
2.2 PC/104 module stacks 5
2.3 PC/104 and PC/104-plus 6
2.4 PC/104 CPU modules 6
2.5 Equipment used in this project 6
2.5.1 CPU module 6
2.5.2 Data acquisition module 7
2.6 Typical setup 8
3 Software design and implementation 9
3.1 Real-time systems 9
3.1.1 Real-time control systems 9
3.1.2 POSIX 10
3.1.3 QNX 10
3.2 Components of a real-time system 11
3.2.1 Processes and threads 11
3.2.2 Real-time scheduler 11
3.2.3 Communication through shared variables 11
3.2.4 Communication through message passing 12
3.3 System overview 12
3.3.1 Inertial measurement 13
3.3.2 Attitude control 13
3.3.3 Propulsion control 14
3.3.4 Data storage & communication 15
3.4 Implementation of lower layer components 15
3.4.1 Data acquisition software driver 15
3.4.2 Servo control interface 19
3.4.3 Propulsion control interface 19
3.4.4 Communication protocol 21
3.5 Implementation of higher layer components 21
3.5.1 Thread priorities 23
3.5.2 Synchronization 24
3.5.3 Proposition for a final design 24
3.6 Remote control and monitoring 25
3.6.1 The MATLAB Graphical User Interface 26
3.7 Chapter summary 28

4 Rocket dynamic analysis 29
4.1 Coordinate system 29
4.2 Equations of motion for a rocket moving in the plane 29
4.3 A simple attitude control system 30
4.3.1 PID controller 31
4.3.2 State-space controller with integral action 31
4.4 Rocket motion in three dimensions 33
4.4.1 Equations of motion for a rigid body 33
4.4.2 Lagrange's equations for moving coordinates 33
4.4.3 Moments of inertia 34
4.4.4 Equations of motion for a constant mass system 35
4.4.5 Equations of motion for a variable mass system 36
4.5 Euler angles 37
4.5.1 Choice of coordinate set 38
4.5.2 Resolving the Euler angles in flight 38
4.5.3 Relationship with the IMU 38
4.6 Dynamic modelling using MATLAB Simulink 39
4.6.1 Moments of inertia estimator 39
4.6.2 Angular velocity estimator 41
4.6.3 Assembly of model 41
4.6.4 Model simulations and results 42
4.7 Attitude control system 43
4.8 Chapter summary 47
5 Summary and conclusions 49
5.1 Future work 49
5.1.1 Inertial measurement 49
5.1.2 Propulsion control 49
5.1.3 Attitude and guidance controller 49
5.1.4 Parachute release control 50
5.2 Conclusions 50
References 51
A Analogue to digital conversion 53
A.1 Range, accuracy and resolution 53
A.2 Digital representation 53
B Implementing a PID controller 55
B.1 Limiting the derivative 55
B.2 Anti windup 55
B.3 Implementation 56
C Switch control circuitry 59
D Using QNX 6.2 61
D.1 Mini how-to 61
D.1.1 Often used console commands 61
D.1.2 Changing permissions for a file to become executable 61
D.1.3 Automatically start programs at system boot 61
D.1.4 Installing QNX 6.2 on the PC/104 using the desktop computer 61
D.1.5 Disabling Photon boot 62
D.1.6 Setting up the network card 62
D.2 Software development in QNX 63
D.2.1 Programming in QNX 64
D.2.2 Compiling C programs 64
D.2.3 Using a remote terminal to access a QNX computer 64
D.2.4 Copying files between two computers using ftp 65

E Software framework for autonomous vehicles 67
E.1 File structure 67
E.2 Threads and the C language 67
E.2.1 Thread overview 68
E.3 Data interchange 69
E.4 Platform communication 69
E.4.1 Communication protocol 69
E.4.2 Type IDs 70
E.4.3 Type ID interpreter 70
E.4.4 Thread synchronization 71
E.4.5 Message passing 71
E.4.6 Remote command relaying 72
E.4.7 Sending measurement data over the TCP/IP connection 72
E.5 Remote platform control 73
E.5.1 QNX and Linux client 73
E.5.2 Windows client 74
E.5.3 Windows client communication model 74
E.5.4 Communication link safety protocol 74
E.5.5 The GUI interface 75
E.6 Tutorial – Using the MATLAB client program 75
E.6.1 Basic operation 75
E.6.2 Extending the GUI with a servo controller 76
E.6.3 Expanding the GUI with new measurement displays 79
E.6.4 Controlling a solenoid valve 80
F Data acquisition software driver 83
F.1 Starting the driver 83
F.2 Controlling the DAQ-card 84
F.2.1 DEVCTL_SETCHANNEL 84
F.2.2 DEVCTL_SETPACER 85
F.2.3 DEVCTL_SET_TRANSFER_RATE 85
F.2.4 DEVCTL_DMA_INIT 85
F.2.5 DEVCTL_ATTACHFILTER 86
F.2.6 DEVCTL_AD_START 86
F.2.7 DEVCTL_AD_STOP 86
F.2.8 DEVCTL_SET_READMODE 87
G Source code 89
G.1 dmadef.h 90
G.2 IDList.h 91
G.3 ioports.h 93
G.4 pulsecodes.h 94
G.5 servocontrol.h 95
G.6 pcm3718.h 96
G.7 pcm3718.c 99
G.8 netserver.c 116
G.9 netclient.c 129
H Matlab Code 133
H.1 findGuidata.m 133
H.2 KeepaliveHandler.m 133
H.3 ConnectionstatusHandler.m 134
H.4 DatarecvdHandler.m 134
H.5 rocketclient.m 135

1 Introduction

The Robotics and Mechatronics Research Laboratory (RMRL) is investigating the possibility of launching an autonomous hybrid rocket. The investigation consists of several studies and smaller projects, mostly performed by final year students at Monash University.

The project presented in this report investigates the onboard computer and attitude control system. Before autonomous algorithms can be constructed, problems such as choosing the appropriate operating system for the onboard computer as well as low-level control of underlying hardware have to be solved.

RMRL has several ongoing projects in the field of autonomous flying vehicles. Besides the hybrid rocket, it is worth mentioning another project investigating a similar control system for a small-scale helicopter. Most of the practical solutions apply to both of these projects and several of the results and even the software components can be reused. This study, however, deals with rockets, which thus will be the focus of this report.

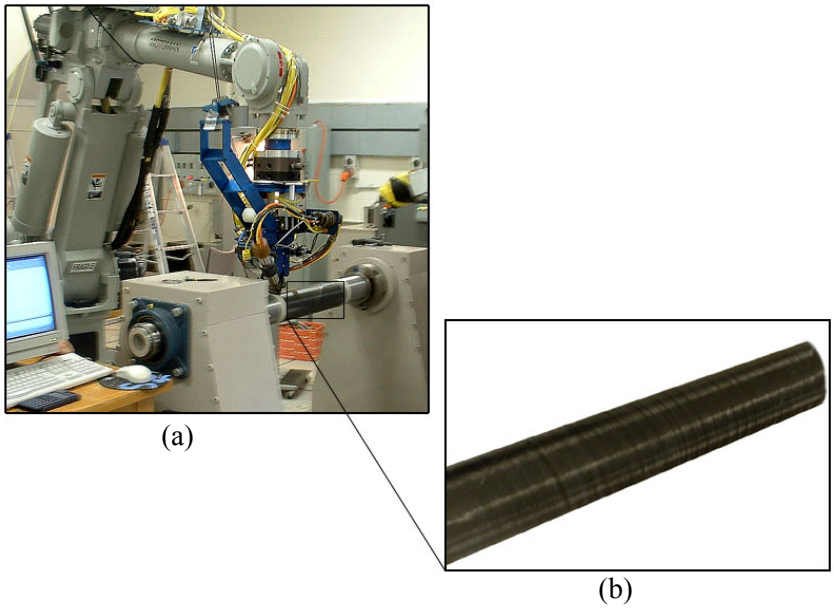
1.1 Rocket modules

The rocket will consist of several individual parts or modules. The main modules are:

1. Structural, such as casing and mounting
2. Hybrid rocket motor with propellant and oxidizer
3. Inertial Measurement Unit (IMU)
4. Attitude and guidance system
5. Power supply
6. Parachute

1.1.1 Rocket casing

The casing of the rocket will consist of a lightweight fibre structure. A group of students within RMRL are conducting research in the field of autonomous fibre placement. The work is based on a robot arm, which automatically places fibres in desired patterns. Layers of carbon fibre and epoxy resin are used for forming different geometrical shapes. The robot can be seen in Figure 1.1(a) in the process of placing fibres onto a cylinder. One such cylinder (with the appropriate dimensions) can be used as rocket casing.



(a) (b)

Figure 1.1: (a) robot arm placing fibres onto cylinder; (b) close-up of fibre structure**.**

1.1.2 Hybrid rocket motor

In a solid propellant rocket motor, the propellant and oxidizer are placed in the same container. Both are shock sensitive and once fired, there is in principle no means for controlling the thrust rate. These problems are overcome when using liquid bipropellant motors instead, which require separate storing of the propellant and oxidizer, resulting in a complex and expensive system.

The hybrid rocket solution utilizes a solid propellant with a liquid oxidizer. It eliminates the shock sensitivity associated with high performance solid propellants by separating the fuel and oxidizer until the firing of the motor. The thrust is terminated with the flow of the liquid oxidizer. The hybrid rocket motor is simpler than a liquid bipropellant rocket motor, but not as simple as a conventional solid propellant rocket motor.

The laboratory rocket motor; seen in Figure 1.2 is developed at the Laboratory for Turbulence Research in Aerospace & Combustion (LTRAC), also located at Monash University. The project is a joint collaboration between the LTRAC and RMRL laboratories.

The hybrid rocket motor uses a propellant consisting of paraffin (wax) with carbon powder. The carbon powder provides an even burn rate. The solid fuel grain in a hybrid rocket motor improves the mechanical properties compared to a solid propellant, because of its lower content of solid oxidizer.

Rocket operation follows a sequence of steps, first using a glow plug in order to ignite a mixture of propane and oxygen inside the rocket combustion chamber. A high pressure solenoid valve is then opened, releasing a flow of oxygen (the oxidizer) into the chamber. The oxygen and the propellant is then consumed inside the motor, thus creating the desired thrust.

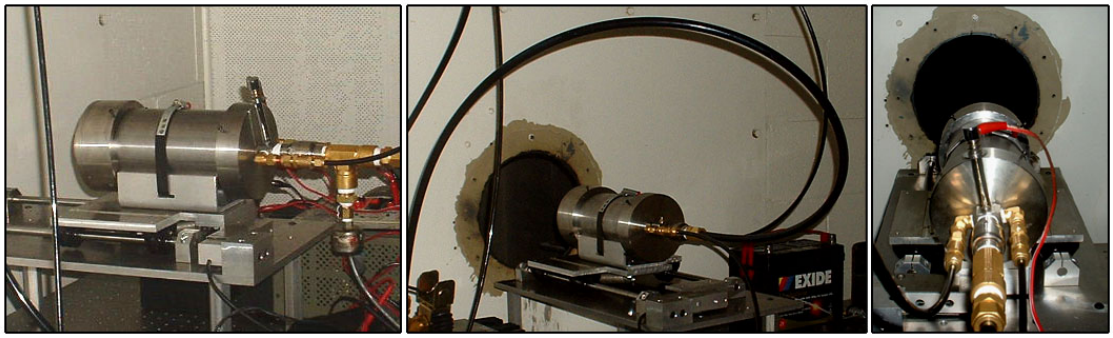


Figure 1.2: Test-setup for hybrid rocket motor. The photo to the far right shows the connections for the propane and oxygen supplies**.**

1.1.3 Inertial measurement unit

In order to navigate the rocket, an Inertial Measurement Unit (IMU) is employed. The IMU consists of accelerometers and gyros. The accelerometers are used for measuring acceleration, velocity and position while the gyros are used for measuring the attitude (rotation) of the rocket. Prior research has been performed in this area at RMRL, and a functioning unit is expected to be ready for testing in the near future.

1.1.4 Attitude and guidance system

The attitude control and guidance of the rocket will be performed by an onboard computer system. The onboard computer will also take care of data acquisition and data handling. The system relies on the IMU to gain position and attitude information. In flight the rocket will be manoeuvred by movable fins on the rocket body (using them as rudders).

1.1.5 Power supply

Electrical power is essential. The onboard computer as well as various subsystems such as the IMU and solenoid valves for rocket motor control requires electrical power (at different

voltages). One or several batteries should be able to deliver the required amount of energy during operation, which in the case of a rocket is not very long (not more than a couple of minutes). It is expected that the necessary study of the electrical requirements is performed in due time and we will therefore not discuss the matter further in this report.

1.1.6 Parachute

Once the fuel inside the rocket has been consumed, the rocket has to return safely to ground. This will require a parachute to be released after burnout. The details of how this is to be performed will be taken care of at a later stage.

1.2 Problem summary

The onboard computer will act as a central unit controlling all the modules inside the rocket.

In general we need the onboard computer for performing the following tasks:

1. **Propulsion control** – Ignition steps and thrust control of the hybrid rocket motor.

2. **Data acquisition** – Gather data from the inertial measurement unit and convert measured values into positions, velocities and attitude.

3. **Attitude control** – Use the external fins to manoeuvre the rocket, based on a decision or feedback algorithm, using data from the inertial measurement unit.

4. **Data storage and communication** – Gather onboard data such as velocity, positions, tank pressures and other critical parameters, either by storing them onboard, or by transmitting them over a communication link.

We can restate these tasks in a simpler form by only concentrating on the central parts. If we solve each of these individual tasks it should be possible to fulfil the requirements stated above.

1.2.1 Propulsion control

The rocket motor is ignited by the opening and closing of solenoid valves, according to a given pattern. The important question is: How do we control a solenoid valve using software in the onboard computer? A similar question can be stated for the switching of the glow plug.

1.2.2 Data acquisition

The electrical signals generated by the components of the IMU need to be measured and converted to a digital format. The question is here: What is needed in order to make the measurements available to the onboard computer in a format usable by the software?

1.2.3 Attitude control

The manoeuvres of the fins by the attitude control system will be controlled through small servos placed inside the rocket. The question is: How do we actuate the servos? Another important question is: How should we design and implement the control algorithms?

1.2.4 Data storage and communication

We need to find a way of storing and/or transmitting the data collected for observation and post processing during flight operation.

1.3 Report layout

This report is divided into two more or less individual parts or investigations. First, there is a part on applied technical issues, where we discuss the structure of the software running the onboard computer. The software structure forms a low-level programming platform, or framework, for further development of the rocket control software. The developed framework uses real-time programming features in order to attain as high system performance and reliability as possible, which is imperative for flight control systems. The discussion can be found in Chapter 3, where the issues of propulsion control, data acquisition and data storage/communication will be investigated.

The second part of this project is more theoretical. Using feedback control theory and theory of body dynamics, a physical model of the rocket has been formulated in order to analyze the dynamic behaviour of the rocket. The model can be used as a starting point for the creation of a feedback system controlling the rocket attitude in flight. This is discussed in Chapter 4.

In Chapter 2 the onboard computer used in this project is introduced. Chapter 2 should be read before Chapter 3.

The Appendices are structured as follows:

Appendix A: A brief introduction to analogue to digital conversion. Understanding of the basic concepts will help in the the discussion of data acquisition in Chapter 3.

Appendix B: Explanation of some common pitfalls when implementing feedback controllers in programming code. This information can be used for implementing the attitude controller in a future project.

Appendix C: Circuit diagrams of the external electronics used by the onboard computer in order for it to work as propulsion controller.

Appendix D: Insight into the use of the QNX operating system. It contains some useful commands and hints often needed when using the OS.

Appendix E: Explanation of the implementation of the software code developed for this project. Here also a programming tutorial is found, explaining how to use the software framework.

Appendix F: Documentation of the software driver developed for the data acquisition module.

Appendix G: The programming code written for the QNX OS.

Appendix H: The Matlab code for the communication solution, which we will discuss in Chapter 3 (also mentioned in the tutorial found in Appendix E).

2 Setup

In order to perform attitude and guidance control of the hybrid rocket, we need an onboard computer system, which must be both powerful and small enough to fit inside the rocket. The computer will use an inertial measurement unit for making navigational measurements. This requires a data acquisition module to be installed in conjunction with the onboard computer. The data acquisition module will also be used for monitoring onboard data, such as temperature and pressure inside the rocket motor. In order to meet these requirements in a cost efficient manner, a PC/104 computer solution is used.

2.1 What is PC/104?

The PC architecture has become an accepted platform far beyond desktop applications. Dedicated and embedded applications for PCs are used as controllers within laboratory instruments, communication devices, and medical equipment, just to mention a few examples. The standard PC form-factor, however, and its associated card cages and backplanes are too bulky (and expensive) for most embedded control applications. In response to this the industry developed a new form-factor named PC/104. It offers complete hardware, and software compatibility with the PC architecture, but in ultra-compact form. Although PC/104 modules have been manufactured since 1987, a formal specification was not published until 1992 [PC104].

The key differences between PC/104 and the regular PC/AT architecture (IEEE P996) are:

- **Compact form-factor** – Size reduced to 3.6 x 3.8 inches (9.1 x 9.7 cm).
- **Self-stacking bus** – Eliminates the cost and bulk of backplanes and card cages.
- **Reduced connector size –** Rugged and reliable 104-pin contact male/female headers replace the standard PC's edge card connectors (hence the name PC/104).
- **Relaxed bus drive (4 mA) –** Lowers power consumption (1-2 Watts per module) but also reduces the maximum number of components on the bus.

PC/104 offers all the advantages in the standard PC configuration, including the use of existing peripheral equipment (hard disk drives, keyboards, monitors etc.) as well as standard PC operating systems. It is thus possible to reuse software solutions developed for standard PC applications, saving development time and costs.

2.2 PC/104 module stacks

PC/104 modules are self-stacking. Stacked modules are spaced 0.6 inches (1.5 cm) apart. The three module stack shown in Figure 2.1 measure just 3.6 by 3.8 by 2 inches (9.1 x 9.7 x 5.1 cm).

PC/104 vendors today offer hundreds of different “off the shelf” modules, meeting even the most specific demands. Examples of existing PC/104 modules are CPU modules, data acquisition modules, I/O modules and network modules.

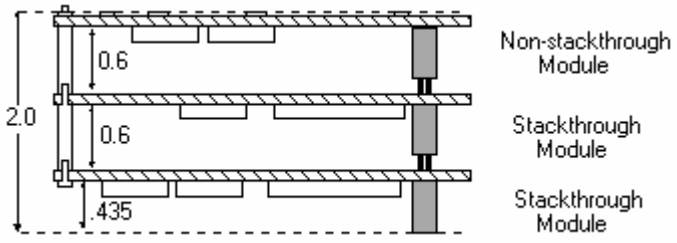


Figure 2.1: PC/104 module stack.

2.3 PC/104 and PC/104-plus

The PC/104 uses the ISA bus as module-to-module bus. The ISA bus is rarely seen in the standard PCs of today due to its speed limitations.

In 1996 the new PCI bus was introduced to the PC/104 community under the name PC/104-plus. In order to retain backward compatibility with the PC/104 bus, it was decided that the new bus should not replace, but extend the PC/104 bus (the 104 pin ISA bus). Thus the same stack can contain both PC/104 and PC/104-plus modules. PC/104-plus modules offer an extra 160-pin stackable PCI bus connector.

2.4 PC/104 CPU modules

All PC/104 solutions start with a CPU module (also known as Single Board Computer, SBC). A PC/104 CPU module is basically an entire PC fitted onto the PC/104 form-factor (9.1 x 9.7 cm), which introduces some limitations. The most important consequence is that the PC/104 CPU modules lag some years behind the frontline PC market.

Apart from offering connection sockets for standard hard disk drives and floppy drives, most CPU modules also offer some solution for embedded memory cards, such as CompactFlash® (or other memory solutions found in today's growing digital camera market). This is ideal for embedded solutions, where space is of major concern.

2.5 Equipment used in this project

This project utilizes two PC/104 modules; one CPU module and one data acquisition module. Both modules were purchased from Advantech Co. Ltd.

2.5.1 CPU module

The CPU module is of model PCM-3350. It has the following specifications:

- NS Geode GX1 300MHz CPU (586/Pentium compatible)
- Up to 128 MB system memory (SDRAM)
- Onboard VGA
- One CompactFlash socket
- Built-in enhanced IDE hard disk drive interface
- Onboard PS/2 keyboard/mouse connector
- Two serial ports (RS-232)
- Onboard 10/100Base-T Ethernet interface (RJ45 LAN connector)
- +5 V power supply at 2A

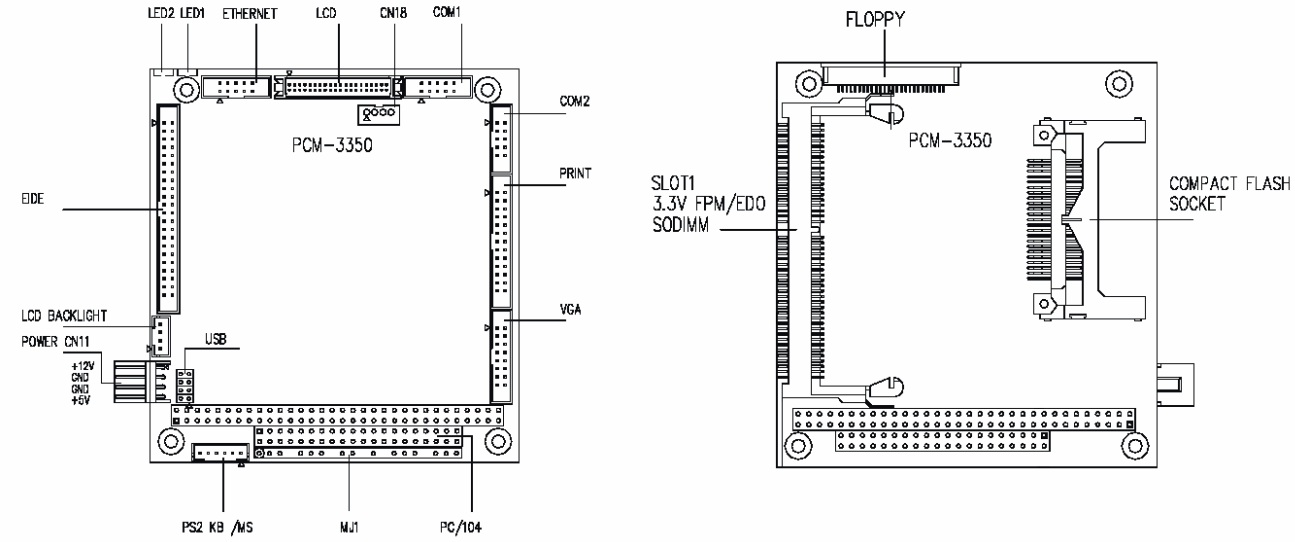


Figure 2.2: Overview of CPU module. Top side (left) and bottom side (right).

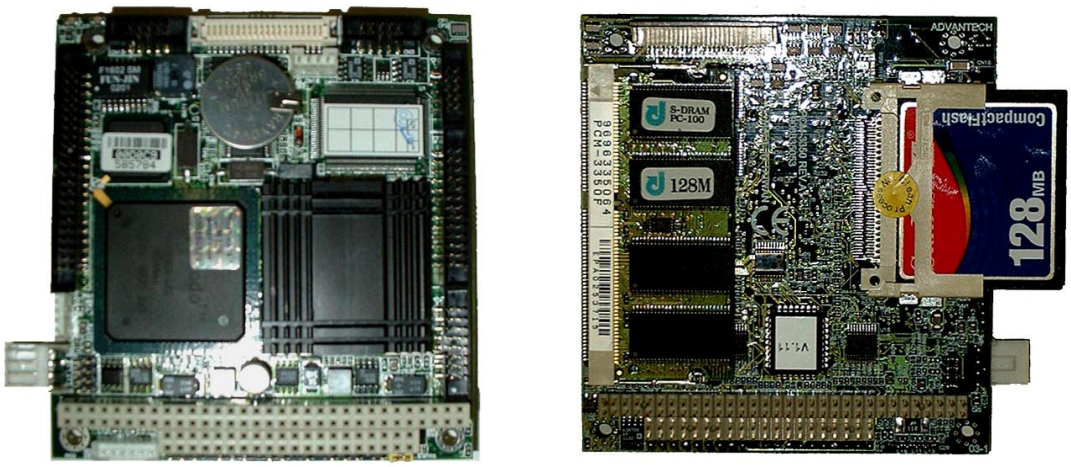


Figure 2.3: Photo of CPU module. Top side (left) and bottom side (right).

The CPU module was fitted with 128 MB SDRAM and one 128 MB CompactFlash memory card (used as hard disk drive).

2.5.2 Data acquisition module

The data acquisition module is of model PCM-3718H. It offers the following features:

- 16 single-ended or 8 differential analogue jumper selectable input channels
- 12-bit A/D converter, up to 100 kHz sampling rate with DMA transfer
- Software programmable gain value for each input channel
- Software selectable input range for each input channel
- Two 8-bit digital input/output channels, TTL compatible
- Flexible triggering options: software trigger, programmable pacer trigger and external pulse trigger
- Data transfer by program control, interrupt handler or DMA
- Power consumption +5V@ 180mA (typical)
- Allowable input ranges (software programmable):
  Bipolar: ±10V, ±5V, ±2.5V, ±1.25V, ±0.625V
  Unipolar: 0 – 10V, 0 – 5V, 0 – 2.5V, 0 – 1.25V

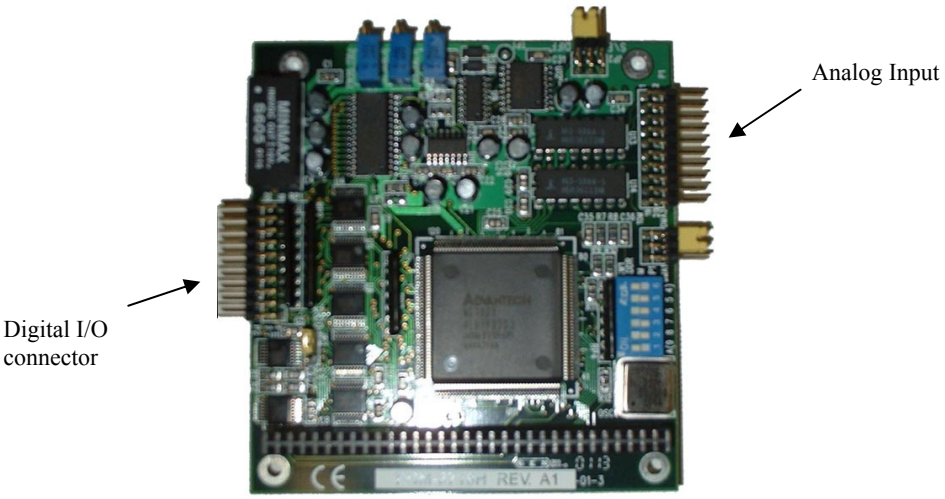


Figure 2.4: Data acquisition module (PCM-3718H).

The PCM-3718H module is placed on top of the PC/104 CPU module using the PC/104 bus. It does not require external power connectors. It draws power directly from the bus.

2.6 Typical setup

Figures 2.5 and 2.6 shows typical setups used during the project. Figure 2.5 shows both modules installed together, where they have been placed in a cardboard holder in order to prevent damage. Once deployed, the dual-card module will be fitted inside a specially manufactured cage holder inside the rocket. Figure 2.6 shows the system installed with most of the external connectors, which may be used for connecting to peripheral equipment, such as keyboard, monitor, printer and network.

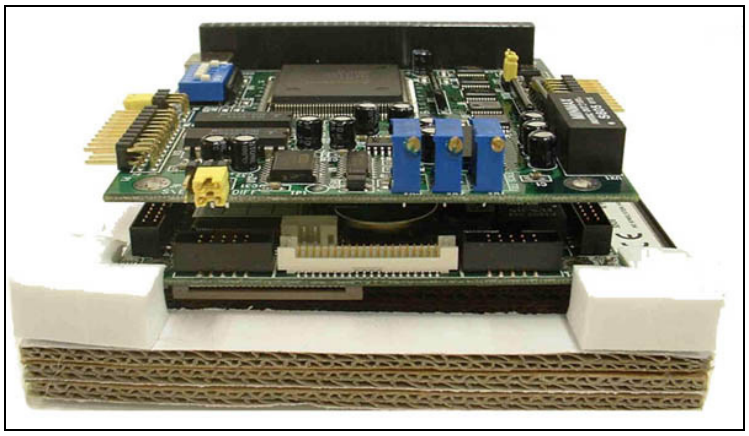


Figure 2.5: PC/104 computer unit without external connectors.

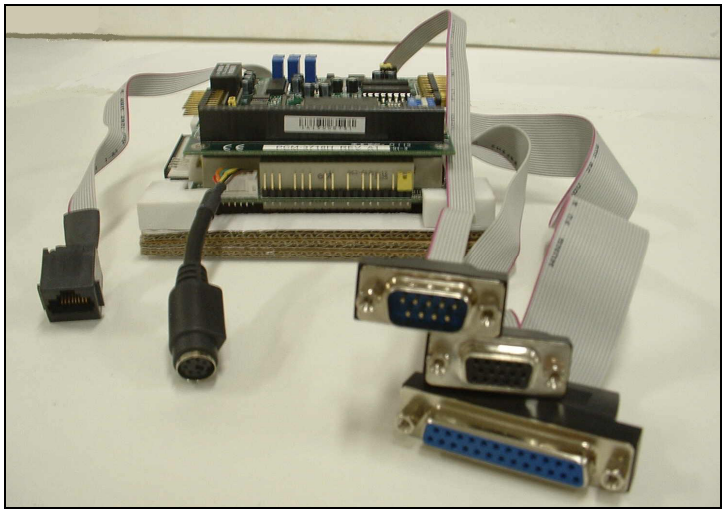


Figure 2.6: PC/104 computer with most of its external connectors.

3 Software design and implementation

The rocket control system puts special requirements on the computer language and operating system. In this chapter, we begin by explaining real-time systems and programming.

Real-time software solutions form an important basis for this project. Unfortunately real-time programming and analysis is not a subject easily explained. It is not possible to cover more than the basics in this report. A reference for a complete introduction is the book by Burns & Wellings [BW97], which is warmly recommended.

3.1 Real-time systems

A real-time system is any information processing system which has to respond to externally generated input stimuli within a finite and specified period (often called *deadline*).

- The correctness depends not only on the logical result but also the time it is delivered.
- Failure to respond is as bad as a faulty response.

If the computer is a component in a larger engineering system, it is generally called an *embedded computer system* [BW97].

3.1.1 Real-time control systems

A control system may be divided into three subsystems:

- Measurement
- Feedback control loop
- Actuation of hardware

Real-life systems are continuous. This often requires the control system to be as closely continuous as possible. It is often desirable to make measurements, execute the feedback control loop and perform actions on the system at the same time. Since a computer only can perform one task at a time, three computers would be needed in order for the three tasks to be performed simultaneously, which of course, is not economically feasible. Instead, methods have been invented, which allow computers to perform several tasks simultaneously using only one central processing unit (CPU). The trick is to switch the task currently performed so fast that it appears that the computer is performing all three tasks simultaneously. This is the general idea behind real-time systems.

Before proceeding, we need to distinguish between four types of *real-time*:

- **Soft real-time** – systems where deadlines are important but which will still function correctly if deadlines are occasionally missed, e.g. data acquisition systems. Common operating systems such as Windows and Linux also fall into this category.

- **Firm real-time** – systems which are soft real-time but for which late delivery of service is detrimental, i.e. any response received after a missed deadline is considered to have no value. This is in contrast to a soft real-time system where a late response still holds some valuable information.

- **Hard real-time** – systems where it is imperative that responses occur within the required deadline, e.g. flight control systems.

- **Real real-time** – systems which are hard real-time and where the response times are very short, e.g. Missile guidance systems.

In this report, we will concentrate on hard and real real-time systems for obvious reasons.

3.1.2 POSIX

In early days, only sequential programming languages were available. Examples are Pascal, C and FORTRAN. Programs written in these languages have a single thread of execution. They start executing in some state and continue executing one instruction at a time, until the program terminates. Today there are a number of so called *concurrent* programming languages such as ADA, Occam2 and Modula. These languages allow the programmer to write code that, if possible, executes concurrently on the computer [BW97].

An alternative approach for achieving concurrency is to use a sequential programming language and an operating system supporting real-time execution. All modern operating systems support such features. One such real-time solution has been standardized under the name "POSIX".

POSIX is a set of international standards defining the behaviour of a conforming application. POSIX is not itself an operating system but rather defines the interface between the application and the operating system. For this to work it is essential that the operating system implementers offer the behaviour defined by the POSIX standard.

A strong feature of POSIX programs is that it is possible to transport source code between any POSIX-compliant operating system, such that similar behaviour is experienced on any such system after compilation.

Besides supporting interfaces for real-time facilities, POSIX also defines interfaces for file handling and security, just to name a few tasks. This report, however, will deal with the real-time facilities only. The POSIX interface is written using the standard programming language C.

3.1.3 QNX

While all operating systems of today have some kind of real-time capability, it is important to classify them as being either *hard* or *soft* real-time operating systems. A UNIX or Windows system for example, may be considered real-time in the sense that a user may expect a response within a few seconds after entering a command (e.g. clicking the mouse-button). Fortunately it is usually not a disaster if the response is delayed or absent. Systems of this kind use soft or firm real-time and can be distinguished from those using hard or real real-time, where failure to respond can be considered just as bad as a wrong response. For example, a missed deadline in a flight control system of a combat aircraft may be catastrophical [BW97].

A number of hard or real real-time operating systems exist. Most of them are commercial, but some also exist in non-commercial versions. The operating system used for this project is the non-commercial version of QNX 6.2 developed by QNX Software Systems Ltd (QSSL). It is a POSIX-compliant OS and has support for modern PC architectures.

QNX uses a micro-kernel which guarantees real real-time support. A micro-kernel is a lightweight OS kernel which gives a small memory trace, thus suitable for embedded applications where computer memory and hardware performance is limited. QNX is a scalable OS where support for different services is plugged into the kernel on demand, such as networking, file system or graphics. This is also suitable for an embedded solution since the size of the system may be customized in order to fit particular needs.

3.2 Components of a real-time system

A typical real-time implementation utilizes a number of components in order to work. Besides functions for maintaining concurrent execution it requires mechanisms such as synchronization, which is used for preventing data corruption and for maintaining execution stability.

3.2.1 Processes and threads

Processes and threads are used in order to allow concurrent execution. Before real-time operating systems were introduced, only one program at a time could be executed. This is encountered in console-like environments where a command or program is executed one at a time in order to produce results. This is called sequential execution as opposed to concurrent execution, in which a large number of commands or programs runs simultaneously. Each program running is in real-time terminology called a *process*. Each process has its own memory space and only that process can change the content of its own memory area (in order to avoid corruption of the stored data belonging to one process, caused by an error or illegal access by another process). In POSIX, the terms *process* and *program* can be used alternatively, since all programs running in the operating system are in fact processes.

Another form of concurrency offered by most real-time operating systems and in some programming languages is something called *threads*. Unlike processes threads are not independent. They are associated with a process and share its memory area with the process. Threads are like lightweight processes, usually smaller in size and less complex. Threads are often used as building blocks in a process for creating a more complex behaviour. A process has always at least one thread.

3.2.2 Real-time scheduler

Actual concurrency is not possible on a single processor since it executes its instructions in a sequence. The effect of concurrent execution can be mimicked by allowing all processes to utilize the CPU for a very short time period, or *timeslot*. It is the task of the scheduler to choose the process to execute during a timeslot. The scheduler is often a process of its own, usually belonging to the kernel of the operating system.

3.2.3 Communication through shared variables

The correct behaviour of a concurrent program is critically dependent on synchronization and communication between processes and threads. A common situation is when a specific action in one thread only occurs after a specific action in another thread. Communication means passing of information between two processes or threads.

Synchronization is an important part of the overall design of a concurrent system. Synchronization can be seen as communication without content, i.e. the synchronization itself represents the required information.

Communication and synchronization is usually based upon either *shared variables* or *message passing*. Shared variables are objects accessed by more than one process or thread. Communication and synchronization is therefore possible by making reference to the variables when appropriate.

The most common form of communication using shared variables is access to a part of the system where only one process/thread is allowed at any one time. Most hardware components fall into this category, for example printers, graphics and I/O devices. This kind of synchronisation is known as *mutual exclusion* and common solutions used for providing this functionality are *semaphores, monitors* and *critical sections*. They will, though, not be treated in this text. For further reading, see the book by Burns and Wellings [BW97].

Another important shared variable mechanism is the *Read/Write lock,* or simply R/W lock. The R/W lock is a mechanism that prevents threads from writing to a specific data area simultaneously (If this happens data will probably be corrupted); thus allowing only one thread to write to the memory area at any one time. If no thread is writing, no changes will be

made to the data, in which case the R/W lock allows read accesses by multiple threads simultaneously, thus increasing data throughput. The R/W lock is the mechanism used in this project for attaining mutual exclusion.

3.2.4 Communication through message passing

The other type of communication is *message-based*. A message-based system requires a sender, a receiver and sometimes a medium for passing the message. This can be a message box or a communication channel.

Customarily, there are three different types of message-based communication:

- **Asynchronous –** The sender proceeds immediately, regardless of whether the message was received or not. Asynchronous communication is found in several operating systems as well as in POSIX.
- **Synchronous –** The sender proceeds only when the message has been received. Synchronous send is found in some programming languages.
- **Rendezvous –** The sender proceeds only when a reply has been returned from the receiver. Rendezvous, also called *remote invocation* is found in Ada and various operating systems including QNX.

This project is based on the message-passing model. Each thread requiring synchronization has its own message-receiving mechanism. Messages are sent between threads in the system allowing synchronization as well as communication.

QNX uses the Rendezvous model for communication, which is a memory consuming and relatively slow form of communication. It is suited for process-to-process communication and not for thread-to-thread synchronization. Fortunately QNX provides a lightweight message called a *pulse*. This project widely uses the pulse mechanism for achieving synchronization.

3.3 System overview

What is it then that we want to achieve with the components described? In order to answer this, let us restate the required tasks mentioned in Chapter 1.

- Data acquisition
- Attitude control
- Propulsion control
- Data storage/communication

Figure 3.1 shows the hierarchy of the control system. The upper layer represents the highest level of abstraction. The components of the upper layer each depend on one component found in the lower layer.

Before complex behaviour such as autonomous control can be achieved, we first need to implement the lower layer of Figure 3.1. This requires vast knowledge of both the hardware in the system as well as the operating system.

The lower layer components act as a framework for the upper layer components. The propulsion control interface should be sufficiently general in order that it may be controlled by different propulsion control algorithms; the communication system (or protocol) should be sufficiently general for it to be used with different data transfer strategies and so on. Note that the lower layer components can be reused in a number of autonomous vehicles such as helicopters, airplanes and, of course rockets. We now investigate the subsystems of Figure 3.1, one at a time.

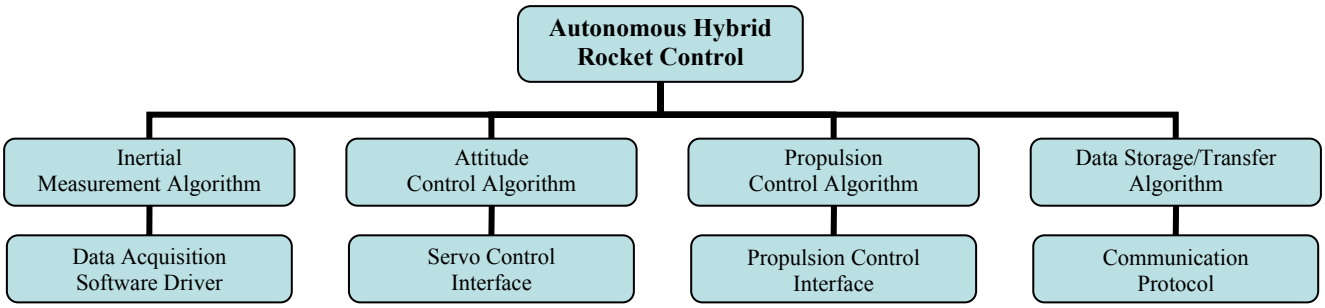


Figure 3.1: Overview of autonomous hybrid rocket control system.

3.3.1 Inertial measurement

Positioning in 3 dimensions is achieved using gyros and accelerometers inside the inertial measurement unit (IMU). The signals from these components can be connected to a data acquisition card and used as input to an iterative software algorithm that outputs useful parameters, such as position, attitude, and velocity. These data are then used as input to the feedback control algorithm (see section 3.3.2).

The data acquisition card can also be used for logging temperatures or pressures inside the hybrid rocket engine, which further justifies the use of general purpose data acquisition software. An overview of the subsystem is shown in Figure 3.2.

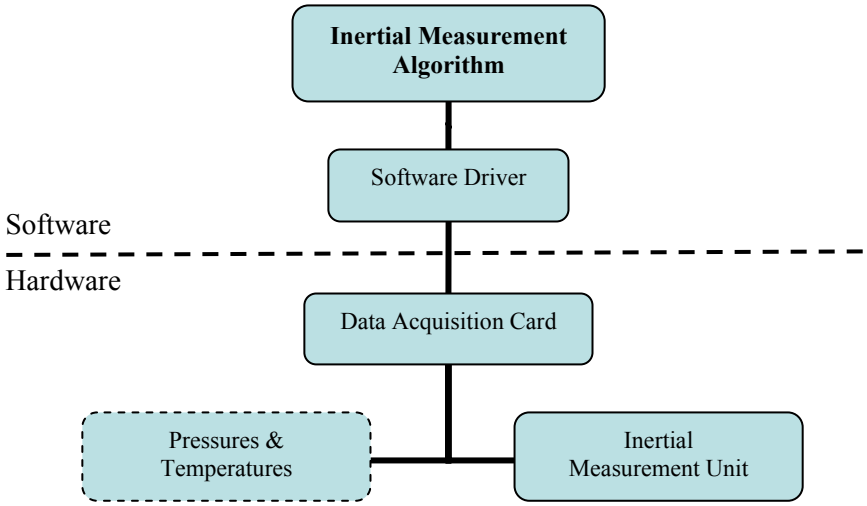


Figure 3.2: Inertial measurement subsystem.

3.3.2 Attitude control

The system controlling the rocket attitude is an important part of the project. Like many feedback control systems, this subsystem may be further subdivided for convenience of analysis; in this case into four parts, as seen in Figure 3.3.

- **Inertial measurement system** – The output from the inertial measurement algorithm mentioned in the previous section is used as input to the control system.

- **Feedback controller** – A software algorithm that makes decisions about how the system should be controlled.

- **Actuation system** – Consists of one or two servo controllers, each controlling a small number of servos, which control the motion of the vehicle by fin-stabilization.

- **Vehicle response** – The dynamic response of the rocket itself. In order to design a good control algorithm, a dynamic model of the motion of the rocket is required.

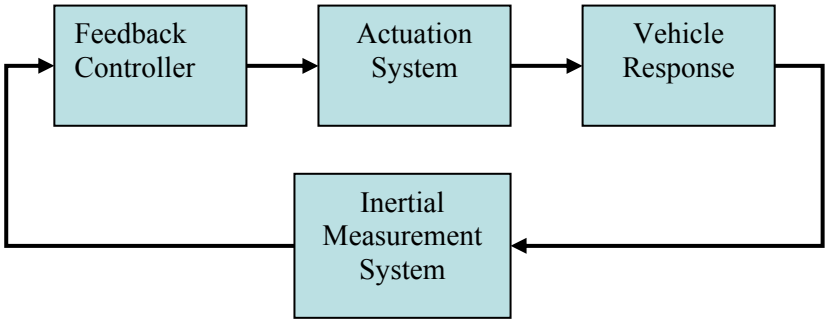


Figure 3.3: Overview of attitude control system.

Figure 3.4 shows the relation between hardware and software in a similar manner as in Figure 3.2.

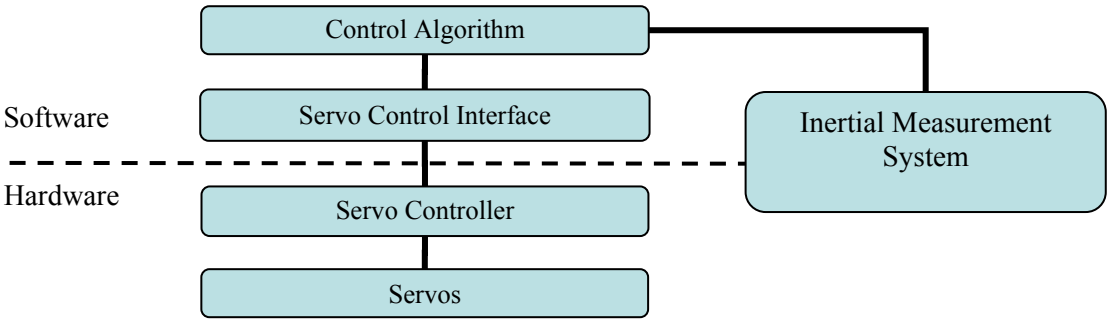


Figure 3.4: Attitude & guidance subsystem.

3.3.3 Propulsion control

The propulsion control system performs opening and closing of the solenoid valves as well as the on/off switching of a glow plug. In future designs advanced thrust control will require partial opening of the valves. This feature will, however, not be implemented in this project.

Valve control cannot be performed directly from the PC/104, but requires electronic components that perform the actual switching. We will discuss later how such control circuits can be constructed, but let us for the moment assume that we have some switching electronics connected to a digital port of the PC/104. Then from a software perspective, the state control of the digital port, along with the switching electronics makes up the propulsion controller. The high-level (upper layer of Figure 3.1) propulsion control algorithm can then be instructed to open and close whatever valves necessary.

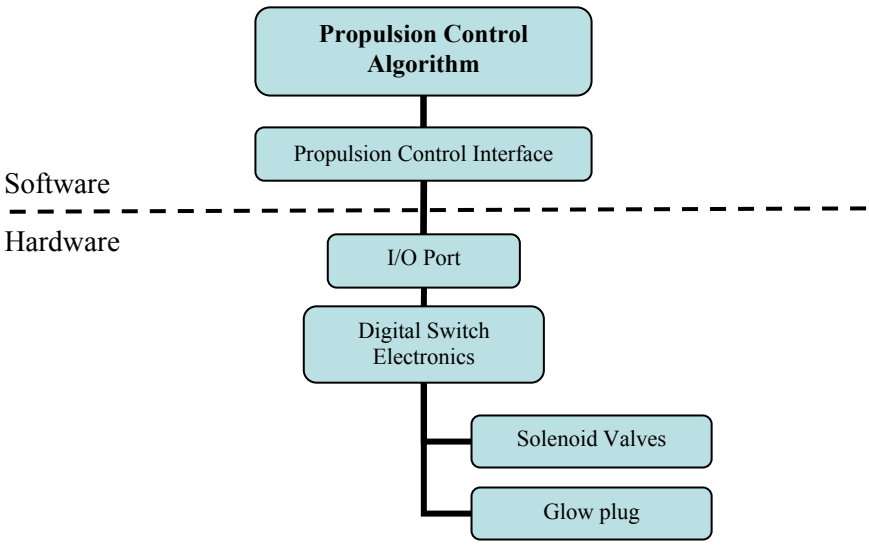


Figure 3.5: Propulsion subsystem.

3.3.4 Data storage & communication

Inertial measurement, propulsion control and attitude control are basically the three subsystems needed for producing a working rocket control system. There are, however, a few more subsystems necessary in a good design.

Maintenance and troubleshooting sooner or later will be required. It is then a distinct benefit to have stored data available for system operation in order to localize both system errors and analyze its performance. A subsystem that might be of interest concerns the communication between the autonomous vehicle and some remote computer acting as "control and observation centre".

Storing data locally on a solid state hard disk is another option. If there is a working communication system, local data logging might not be preferred to the option of storing data in the remote system.

3.4 Implementation of lower layer components

We have briefly discussed the necessary components for a real-time system and what we need in order to implement our particular system. In Figure 3.1, we presented an overview of the software control system. In this section the software implementation is described along with a discussion of the components in the lower layer. The components needed are:

- Data acquisition software driver
- Servo control interface
- Propulsion control interface
- Communication protocol

3.4.1 Data acquisition software driver

An inertial measurement unit requires proper data acquisition routines and hardware in accord. This project concentrates on the data acquisition for two reasons: Firstly because no inertial measurement unit was ready for testing, and secondly that prior research of the data acquisition hardware was inadequate. No software driver for the QNX 6.2 operating system was available, which therefore had to be developed.

3.4.1.1 Software drivers and resource managers

A computer would not be of much use without hardware components connected to it. In the case of a PC/104 computer, it is the PC/104 bus that extends the system. The electrical and

functional properties of the PC/104 bus are equivalent to the standard ISA bus found in desktop PCs.

Computer programs usually never communicate on the bus directly, but needs a software driver, to which the program sends its requests. The driver then interprets these requests and performs the appropriate actions on the card it is assigned to. Unlike operating systems such as Linux, where drivers are tightly connected to the kernel, a QNX driver looks and acts just like any program. In QNX, drivers can be initiated and terminated at any time while the computer is running, which makes it a very flexible OS. In QNX, the name *resource manager* is used instead of software driver. It will be used for the rest of this chapter.

As an instructive example, Figure 3.6 shows how the resource manager developed for the DAQ card interacts with both the card itself and with programs requesting access to the card. The resource manager follows standard QNX driver design methods.

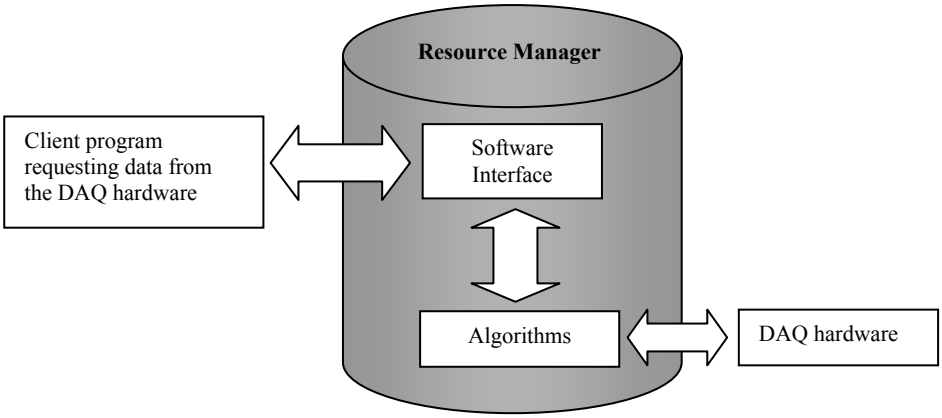


Figure 3.6: Overview of data acquisition resource manager (software driver).

3.4.1.2 Using DMA in data acquisition applications

Direct Memory Access, or DMA for short, is a solution for computer hardware for sending and retrieving data from system memory (RAM). DMA relieves the CPU from dealing with the incoming data sent from the data acquisition card. This means that the DAQ hardware fills a predefined area in the system memory without need for CPU intervention.

3.4.1.3 Memory hierarchy

In order to better understand how the measured data travel from the DAQ hardware to a client (program) using the resource manager, we need to discuss the different memory areas used in the design. A hierarchy is shown in Figure 3.7 and a simple description of how the resource manager operates follows:

1. Create a memory buffer in physical memory (RAM), here called “DMA-buffer”.

2. Initiate the DMA controller and supply the memory address of the previously created DMA-buffer.

3. Initiate A/D conversion (The A/D converter starts filling the DMA-buffer with data).

4. The resource manager copies the most recent data to a shared memory area when the client requests data. The shared memory area is accessible from both the resource manager and the client. The client now has a copy of the data sent from the ADC (A/D converter) and this completes the transfer cycle.

Recall from section 3.2.1, that each program has its own memory area. The resource manager is, in fact, a program running in the background. This means that the client cannot access the DMA-buffer directly, since it belongs to the memory area of the resource manager. Fortunately, the QNX operating system allows programs to share data dynamically using so called *shared memory objects*. By putting data in any such object (named "Exposed data" in Figure 3.7) the measurement data can make its way into the client memory for analysis. This model is very safe and minimizes the risk that malicious code on the client-side corrupts the behaviour of the resource manager (at a cost of a slightly slower system response).

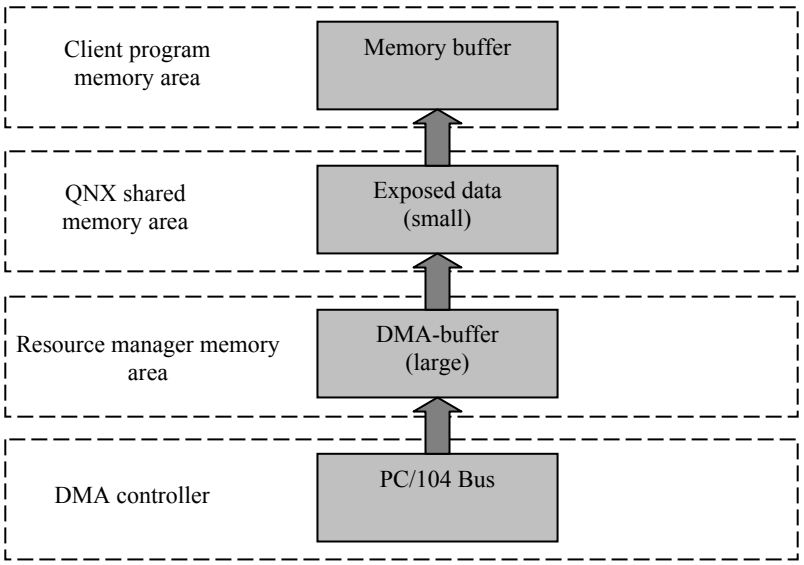


Figure 3.7: Memory hierarchy for data acquisition.

3.4.1.4 Data synchronization

Almost every application using data acquisition requires the stream of data to be synchronized. Consider for example a feedback control application using the signal from an A/D converter as input. The feedback control loop is most certainly synchronized using some timer mechanism (this is where real-time programming is used) and most probably it requests new A/D data every time the loop is executed. This means that data will be requested at the same frequency as the feedback control loop. Thus, the A/D data update frequency inside the client must be at least as high as or higher than the frequency of the feedback control loop.

There are two strategies for synchronizing data transfers from the resource manager:

1. **Asynchronous –** The client acquires data by sending requests to the resource manager. The resource manager then sends the data back to the client (using shared memory objects).
2. **Synchronous –** The resource manager is responsible for the correct timing of the data transfers. The manager notifies the client each time updated data in the shared memory area is present (see Figure 3.7).

The first method is easiest to implement. Unfortunately it can cause unexpected problems. When the client requests data from the resource manager, the manager might not be able to respond directly. Besides transferring data to the client, the resource manager has to react on hardware interrupts caused by the DAQ hardware. This requires some extra work to be performed at specific intervals, which may cause delays and timing errors in the data transfer that cannot be foreseen by the client.

The second method requires more setup and initialization by the client in order to work. The benefit, however, is that it is the resource manager which keeps track of the time for data transfer. Thus, the resource manager can prepare in advance for the transfer of data. This gives a very good timing confidence. This second method has been used in this project.

3.4.1.5 Data transfer & timing

When the analogue converter is working at a high pace, which is normally the case, it is unrealistic for the client to instantly analyze the incoming data. Instead, blocks of data, containing more than one A/D converted measurement, are sent from the resource manager to the client. Thus there are two frequencies that we can vary independently. The first frequency is that of the pacer of the ADC, controlling how many measurements per second the ADC is executing. The other frequency determines how often the client requests data from the resource manager. In order to get a feeling for what this is all about, we demonstrate some simple examples.

Table 3-1 Examples of pacer and transfer frequencies.

ADC frequency	Client frequency	No. of measurements / transfer
10 kHz	1 kHz	10
50 kHz	1 kHz	50
100 kHz	2 kHz	50

The feedback control loop located in the client will most likely never be executing faster than 1 kHz, even in rocket guidance applications.

In the first example of Table 3-1, 10 measurements will be transferred from the resource manager to the client every millisecond. At this instance there is another important matter to keep in mind concerning the ADC: What is the ADC measuring?

The ADC can convert any number from 1 to 16 signals depending on the setup. In the ADC conversions are done sequentially; i.e. it first converts channel zero, then channel one, then channel two and so on until it reaches its limit (which is software controlled). It then restarts the conversion from channel zero again. Investigating Table 3-1 we cannot tell how often a specific measurement channel repeats itself. In the first case, where ten measurements are sent every millisecond, it could mean ten distinct samples from the same signal source, or it might be ten different signals.

We know that the inertial measurement unit uses six signals (three gyros and three accelerometers), if we have only one to five of these signals at a given instance of time, we cannot use them as input to the inertial measurement algorithm; we need all six measurements.

Making analogue measurements it is usually desirable to sample all the signals of interest at least once before starting any analysis, i.e. one full set of measurements is needed. The number of complete sets of measurements required for each update period in the client program thus needs to be estimated

As a counter-example, let us consider the case where only the six channels from the IMU are measured. Further assume that the ADC is running at 50 kHz, and that the inertial measurement algorithm (located in the client program) is running at 1 kHz, generating data-blocks of 50 measurements for every transfer cycle (as in the second case of Table 3-1). Then eight complete sets (8*6=48) are generated, but only the first two measurements from the ninth set are obtained after the first transfer cycle. Hence the question about handling the last two values must be addressed. If they are ignored, it will be necessary also to ignore the first four measurements of the next block since they are part of the same set. Alternatively the last

two measurements from the previous block may be stored and then reconnected with the first values of the next block, thus, in effect, mixing old and new data.

This question does not have a simple answer. The only recommendation the author can give, is to try to avoid the problem altogether. This can be done by selecting frequencies for the ADC and client such that they match the number of channels, and thus only multiples of full sets are generated each time the data is transferred from the resource manager to the client.

3.4.2 Servo control interface

There are many different servo motors on the market, all designed for different applications and different environments. They also offer different control interfaces depending on how accurate the servo movement must be. One servo control interface is the *pulse width modulation* (PWM) which is a simple and low-cost method of servo control. At the time when this project was performed, the final choice of servos and servo controller had not been made. The robotics lab had, though, some PWM controlled servos that could be used for the time being. This section discusses a small software interface controlling these servos. Figure 3.8 shows the servo and the servo controller. A servo controller is a small piece of electronic hardware required for generating the PWM signals.

Using a servo controller for controlling servos is straightforward. The controller can operate up to eight servos at the same time. Control commands are sent to the controller through a serial cable connected between the controller and one of the PC/104 serial communication ports.

The software we need for controlling the servos is simply the code for sending the correct control characters to the serial port of the PC/104. Many features involving serial communication are already implemented in the QNX operating system. High-level programs can easily access the servo controller by using the very small serial interface written as a part of the software framework in this project. The code may be found in Appendix G.

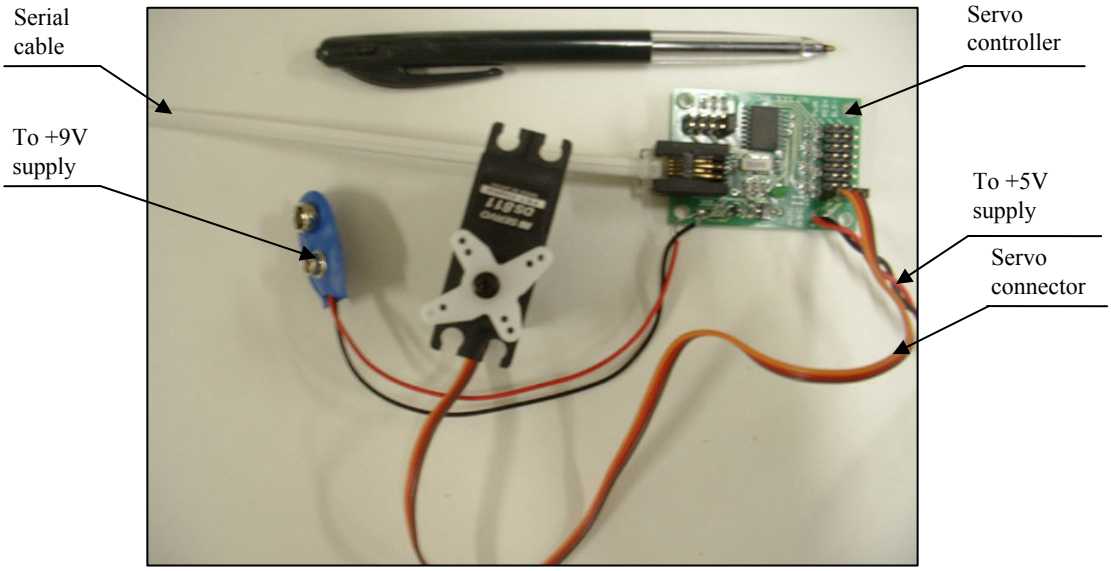


Figure 3.8: Servo and servo controller.

3.4.3 Propulsion control interface

The ignition process is divided into two stages. In the first stage, oxygen and propane flows at a very low pressure past a heated glow plug which ignites the gas. When ignition has been established, the main oxygen supply is opened while the other supplies are closed. The main gas supply flows at very high pressure through the rocket motor, acting as oxidizer for the solid rocket propellant.

Since the two oxygen flows have different pressures, they cannot share the same solenoid valve. Thus we need two oxygen valves, one propane valve and one glow plug. The ignition process is summarized briefly below.

1. Switch on the glow plug
2. Open the propane valve
3. Open the “low pressure” oxygen valve
4. Wait for ignition of the gas mixture
5. Switch off the glow plug
6. Open the main oxygen valve
7. Close the propane valve
8. Close the “low pressure” oxygen valve

Opening and closing the valves are performed using digital switching electronics located between the digital ports of the PC/104 and the solenoid valves and glow plug. Detailed schematics and a description of the switch can be found in Appendix C. The main component is a transistor, controlling some external power source strong enough to open the valves or switching the current for the glow plug. For demonstration purposes, a digital switch was manufactured and tested with one of the solenoid valves. The setup can be seen in Figure 3.9. The parallel port on the PC/104 was used for digital interfacing. This setup, using a 12V power supply for the solenoid valve, worked as intended.

There is a potential problem, however, using the parallel port. Instead the digital I/O ports on the DAQ card ought to be used for propulsion control. This is described in more detail in Appendix C.

The glow plug uses about 3A at almost 1.5V and thus requires some further electronics for converting a 12V (or 5V) signal to the high current 1.5V signal. The implementation of this has not been chosen at the time of writing and requires further investigation.

Once the voltage converter for the glow plug is installed, it should be possible to connect four identical switches (three solenoid valves and one glow plug) to the PC/104 in order to control the entire propulsion system.

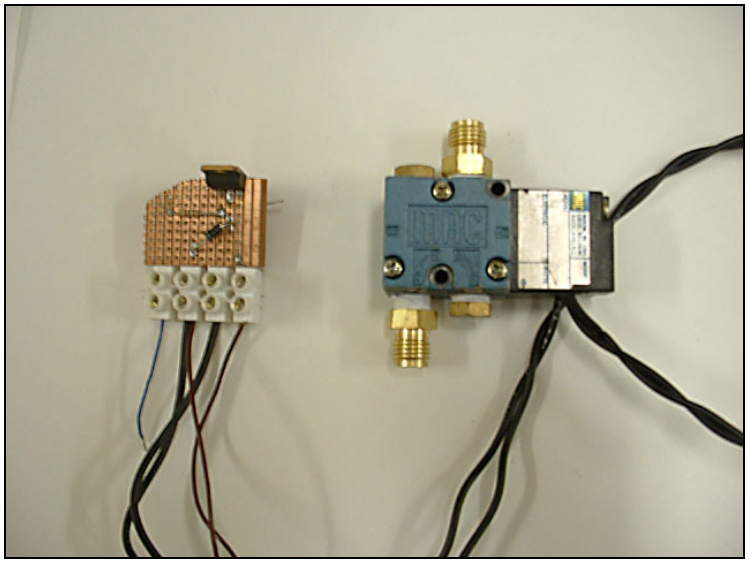


Figure 3.9: Switching electronics (left) and solenoid valve (right).

3.4.4 Communication protocol

We mentioned in section 3.3.4 that communication is not vital for the operation of the rocket (otherwise we would not call it "autonomous"). It is, however, a very useful tool during development and testing, e.g. for finding errors or for tuning system performance. Using a communication system also is of advantage, once the PC/104 and underlying systems are installed in the rocket. Instead of unmounting the PC/104 or its memory, the communication cable may be utilized collecting information, such as stored data.

The communication solution developed for this project uses the TCP/IP protocol for data transfer. The TCP/IP protocol is used today in major communication tasks, such as the Internet. The PC/104 has a built in LAN (Local Area Network) port which is used for all communications.

The LAN connector has several attractive features for communication. Besides support for the TCP/IP transfer protocol, the most important are:

- Fast data transfer (up to 100 Mbit/s)
- Built in error detection and correction for the TCP/IP protocol (no need to develop custom algorithms)
- Vast experience in use with desktop computer applications

A very interesting aspect of LAN is the increasingly growing market for wireless communication, which offers small, cheap and technically advanced solutions such as Wireless Local Area Networks (WLAN). Later it might even be possible to embed a WLAN transceiver in the autonomous flying vehicle, thus offering a cheap form of communication even when the vehicle is in flight.

The TCP/IP protocol is integrated as part of the QNX operating system, which offers much of the implementation we need at no extra cost. QNX supports the standard programming model for network programming (the Berkley model). It is not difficult to implement sophisticated communication solutions, since there are very good references on the subject, both in the relevant technical literature and on the Internet.

3.5 Implementation of higher layer components

The upper layer in Figure 3.1 is only partially implemented. Some components, like the inertial measurement algorithm and the feedback control algorithm have not been implemented at all. This was mostly due to lack of necessary pre-testing of hardware such as the inertial measurement unit.

Nevertheless, we can at least implement the most basic parts of the control software for the PC/104 computer. By representing each component in the upper layer of Figure 3.1 as a real-time thread, we can make a first layout of the program, as seen in Figure 3.10.

Since we are unable to implement inertial measurement algorithms without the IMU we replace the inertial measurement thread with a digital filter thread. A filter is a necessary component in every application that uses analogue measurement data. Its main purpose is to filter out measurement noise. This can be a simple low-pass filter for filtering voltage spikes and other unwanted phenomena encountered in analogue measurement. Most likely it will be necessary to filter out specific frequencies induced by vibration once the rocket is assembled and tested. After the filtering of the incoming data, the inertial algorithm can start execution. The code for the inertial measurement algorithm can either be placed at the end of the filter thread itself or as a separate thread.

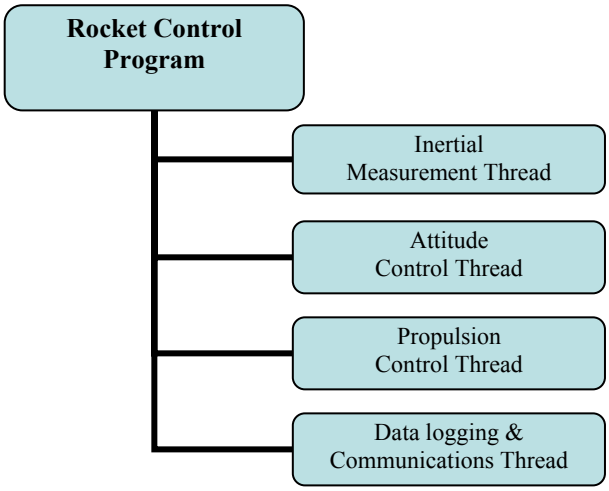


Figure 3.10: First thread layout of control program.

A good design philosophy is to be generous with the number of threads in a program. By using many threads, single threads may be dedicated to performing specific tasks, making the development of code fast and error free. This philosophy has been applied to the Data logging & communications thread. Instead of one thread, we use four threads, each performing a specific part of data gathering and transfer.

The attitude control thread is also a good candidate for being split into multiple threads. Most likely, it will be converted to at least three threads, each stabilizing one of the Euler directions roll, pitch and yaw. We will discuss feedback controllers and Euler rotations in more detail in chapter 4. For the sake of discussion we will stick to one thread only, knowing that it just acts as a placeholder for a more complex system in the final design.

The four threads of the communication system need some explanation. First, in order to achieve what is known as *full duplex*, i.e. sending and receiving data at the same time, we require two threads; one that listens for incoming data over the network ("Listen thread"), and one that sends data over the network ("Send thread").

The properties of the TCP/IP protocol have some drawbacks. One of these concerns the possibility to detect when the computer at one end of the established link has crashed, has become locked up or something similar, i.e. the link is open, but one side of the link has stopped responding. One way to solve this problem is by sending *keepalive* or *heartbeat* data over the link, even when no actual data is being sent. In this way, one side of the link can detect when the other side has stopped responding. This is very important in real-time applications. Suppose the autonomous vehicle keeps a communication link open using TCP/IP during normal operation. If the link is compromised or even severed, then the autonomous vehicle need to know this as soon as possible in order to perform alternative actions. This can e.g. include starting a local logging routine for saving important data to disk that might otherwise be lost while the link is not working. The word *keepalive* is also used in network terminology when talking about a different type of function. Therefore we will refer to this system as *heartbeat*, after the algorithm with that name in the book by Stevens [Ste98], which has inspired the inclusion of this safety mechanism. The heartbeat mechanism uses a separate thread.

The fourth and last thread of the communication system is the data logging. The data logger is an independent thread, monitoring the active state of the system and saving it to disk. As we have mentioned, the data logger is only necessary if no communication link is present. Therefore it is considered to be optional in the sense that it may be turned on or off as the user sees fit. The data logger uses standard C-functions for accessing the file where it saves the data.

Finally, we can draw the new layout of our control program.

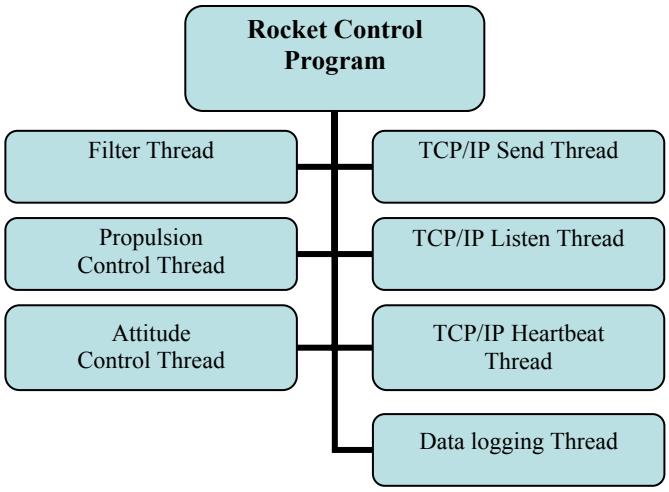


Figure 3.11: Improved thread layout of control program.

3.5.1 Thread priorities

Each thread in a process (or program) has a priority assigned. The priority is represented by a numerical value that contains information for the real-time scheduler about the relative importance of the thread for accessing the CPU. In simple terms, the scheduler selects the thread with the highest priority in order to assign which thread will use the CPU during the next timeslot. If the thread with the highest priority is suspended (e.g. it is waiting for some event to occur), the scheduler will select the thread with the next highest priority. If two threads with the same priority are ready to run, then an arbitrary thread is chosen or, depending on the current scheduling policy, two or more threads with the same priority will be executed using a round robin algorithm, giving equal amount of time to each thread. In QNX, priorities can be any number between 1 and 63 where 1 means lowest priority and 63 is highest. By default, a thread has priority 10.

We will now analyze the choice of priorities for our system in Figure 3.11. The attitude control is the most important part of the system. If it would be pre-empted by some other thread while performing its task, a risk of disaster is inevitable. Therefore we raise its priority to higher than normal, say 15.

Without recent data from the inertial measurement unit, the attitude control thread cannot execute. We cannot, however, raise the priority of the inertial measurement algorithm higher than that of the attitude control thread, and thus risking pre-emption (causing a timing delay of the control algorithm). Therefore we need to choose a lower priority for the inertial measurement algorithm, say 12.

If the attitude control thread should wake up and pre-empt the inertial measurement algorithm it will have to wait until the algorithm has finished executing. This scenario is unacceptable. In other words, we must guarantee that each time the attitude control thread starts executing, the inertial measurement algorithm has completed its task. This can be solved by first performing an analysis of the execution times of each thread in the system. We then decide update frequencies for the measurement and control loops such that their execution windows never coincide.

The other threads in the system can be left at priority 10. The propulsion system can be considered to be more important than the other threads at priority 10. But because of the fact that the thread will only execute a few lines of code (opening/closing valves) it was not considered necessary to raise its priority.

3.5.2 Synchronization

Another matter that needs to be discussed is that of synchronization. We mentioned in section 3.2.4 that we use pulses for almost all communication. Pulses are used for passing events between threads and real-time objects such as timers. All threads that rely on events from other threads or timers possess message retrieval loops, making them wait for some event to occur, e.g. a timer or an event generated by another thread. When such events occur, the acting part sends a pulse to the thread (or threads) that is dependent on the event. A thread waiting for the event, then wakes up and performs the appropriate action. Note that a thread can wait for multiple events simultaneously. If two pulses should arrive at the same time, say from a timer as well as some event generated in another thread, then the priority of the pulse determines the order of action. Pulses have priorities between 1 and 63, exactly the same way as the threads. All threads in Figure 3.11 are event driven (most of them use timers) except the Listen and Heartbeat threads, which both of them listen for incoming data on the network link, in a way making them event driven as well, but they listen for external events rather than internal events.

3.5.3 Proposition for a final design

The layout found in Figure 3.11 was used for this project. As we have previously stated, this layout is not final. The program is only partially implemented and many more hours of work are needed until it will be complete. As a final remark, the thread layout of the complete program will be discussed below.

First, assume that we expand the attitude control thread into three separate threads. Since the three control threads will literally fight over the control of the manoeuvre fins we will probably need a separate thread with the task to coordinate and optimize the servo movements.

Further, the final rocket design will need to deploy a parachute once the rocket is out of fuel. This can be taken care of by a separate thread.

We also need a thread which acts as the “intelligence” or the “autonomous” part of the system. This thread should change the reference values for the attitude controller (i.e. guidance) and give order for ignition of the rocket motor and release of the parachute. During initial test stages, this thread should contain manually written testing manoeuvres. For example:

1. Start rocket engine
2. Wait 5 seconds
3. Perform a 45° roll followed by a pitch change of 20°
4. Turn off rocket engine
5. Release parachute

(In chapter 4 we will explain roll and pitch manoeuvres)

Finally, we must not forget the inertial measurement unit and its corresponding software algorithm, which calls for an additional thread. For simplicity, assume that the rocket does not use network communication, so that we can remove the corresponding threads. The diagram of this final proposition is found in Figure 3.12.

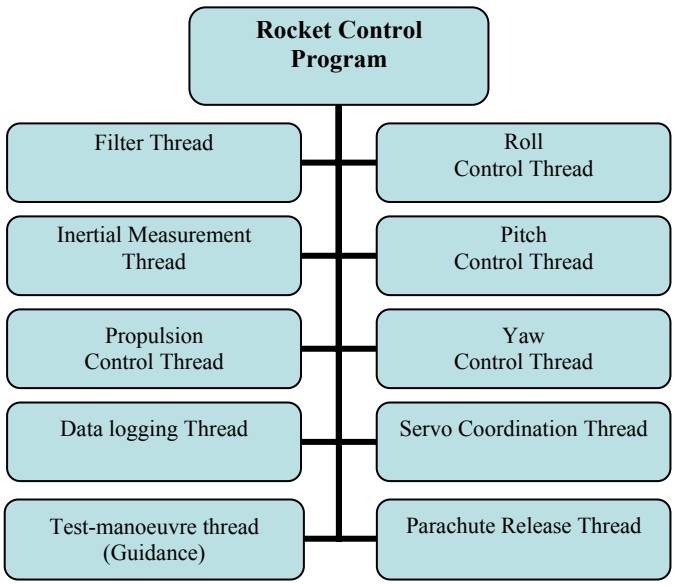


Figure 3.12: Proposition of the final thread layout.

3.6 Remote control and monitoring

Development of the software for the PC/104 computer has almost exclusively been performed using a desktop PC with QNX installed. The compiled programs had to be moved or copied to the PC/104 computer for testing and debugging, because the PC/104 had the required hardware (DAQ card) connected.

The QNX operating system, as any UNIX-style system offers terminal login over a network as a built-in service. This was used extensively during development. Connecting the PC/104 computer over a local area network allows the desktop PC to be used for development as well as for testing, using a terminal connection to the PC/104. This is almost like controlling two computers using only one keyboard and one monitor. It is also possible to transfer the compiled programs from the development computer to the PC/104 for testing, using file transfer programs. The drawback with terminal login is that it only offers a text interface to the remote computer.

After some time we realized that a more sophisticated user interface would be required for controlling and monitoring the functions of the PC/104 computer. What we needed was the possibility to simultaneously monitor multiple parameters and to send control commands or even sequences of commands to the PC/104 while it was running. Connecting a monitor directly to the PC/104 as a graphical interface is out of the question. This would just disrupt the response time of the PC/104 computer since we wish that the system perform as close to real operation environment as possible.

By continuously storing all relevant parameters (time, velocity, attitude and so on), the entire system can be carefully monitored during operation. The stored data can also be accessed later if necessary. In some cases we also need to perform processing of the data such as FFT (Fast Fourier Transform) which can reveal vibration frequencies in the measurements.

We also need to send commands to the PC/104 computer. The control commands sent to the PC/104 could be test commands for use during development. For example, the ignition process could at first be manually controlled by remotely opening and closing the solenoid valves one by one. This approach is recommended before implementing an autonomous algorithm. It would likewise be attractive if the monitoring program had a graphical user interface.

A solution to all of the above is to use MATLAB from The Mathworks Inc. It offers advanced functions for storing and processing data as well as capabilities for displaying graphs of the data. It also offers an easy-to-use graphical user interface builder called Guide.

This builder can be used by anyone familiar with MATLAB for designing graphical user interfaces.

MATLAB, however, comes short on one point; the ability to communicate over the local area network. Actually MATLAB does offer interfaces for this kind of communication, but not versatile enough for handling the requirements for our communication solution.

Fortunately, the author was familiar with COM programming. COM (Common Object Model) allows Microsoft Windows programs to communicate and send data to each other while they are executing. A C++ program was thus written, which took care of all the networking problems that MATLAB was unable to solve. Then by the use of COM technology, the C++ program was used for extending MATLAB with advanced networking capabilities, without compromising its own functionality.

The discussion of COM and network programming under Windows are subjects far too complex to be included in this report. For reference, see the books from Schildt [Sch00], Stevens [Ste98] and Troelsen [Tro00].

What we have achieved by all this is to allow MATLAB running in Windows on a desktop computer to communicate with our program in QNX on the PC/104 computer using the TCP/IP network protocol.

The communication solution described in detail in Appendix E, can be used for communication with any computer platform. There were plans for developing a graphical control program for the Linux operating system as well, but unfortunately this was not allowed within the project time frame. A text version for both the QNX and Linux environments exists and can work as a basis for further development of a graphical client. (Because of POSIX-compliance, the same program compiles both in Linux as well as in QNX.)

3.6.1 The MATLAB Graphical User Interface

Since most of the time spent on this part of the project was used for developing the communication solution and for implementation of the communication protocol, little time remained for the *graphical user interface* or GUI for short.

Now recall that in order to control the rocket we need to be able to open and close solenoid valves and to control servo motors. We also need to monitor internal signals such as those representing velocities and attitude. This was kept in mind when designing the GUI presented in Figure 3.13. It offers a simplified view of what the GUI may look like once it is fully developed. In Figure 3.13, we have a small number of buttons and text boxes. Two buttons take care of the remote connection to the PC/104 computer, and the other buttons may be used either for starting the ADC, for setting a connected servo to a specified position or for controlling the state of a solenoid valve. In addition the GUI displays four A/D conversion channels in some pre-specified unit (e.g. Volts). It also incorporates a text window which informs the user about all communication events (e.g. when the connection is opened or closed). Figure 3.14 shows a corresponding setup of the PC/104 computer used in conjunction with the program.

A quite nice feature of the MATLAB GUI builder is the drag and drop solution for the graphical components. With a mouse-click the new buttons may quickly be added to the GUI without going through the time consuming process of compiling the program.

Once the new buttons have been installed, only a few lines of MATLAB code are necessary in order to add functionality. In most cases, only one line of code is necessary for each new button. The idea is that the communication system and the GUI control should be simple enough for non-experts to use and extend.

Here an interesting option appears. Since we use the TCP/IP protocol for communication, the same protocol used on the Internet, this means in theory that a person using MATLAB (or any COM-enabled program) on a desktop computer can monitor and/or

control the autonomous vehicle from anywhere in the world in real-time (disregarding the effects of transmission delay of the signals over the Internet).

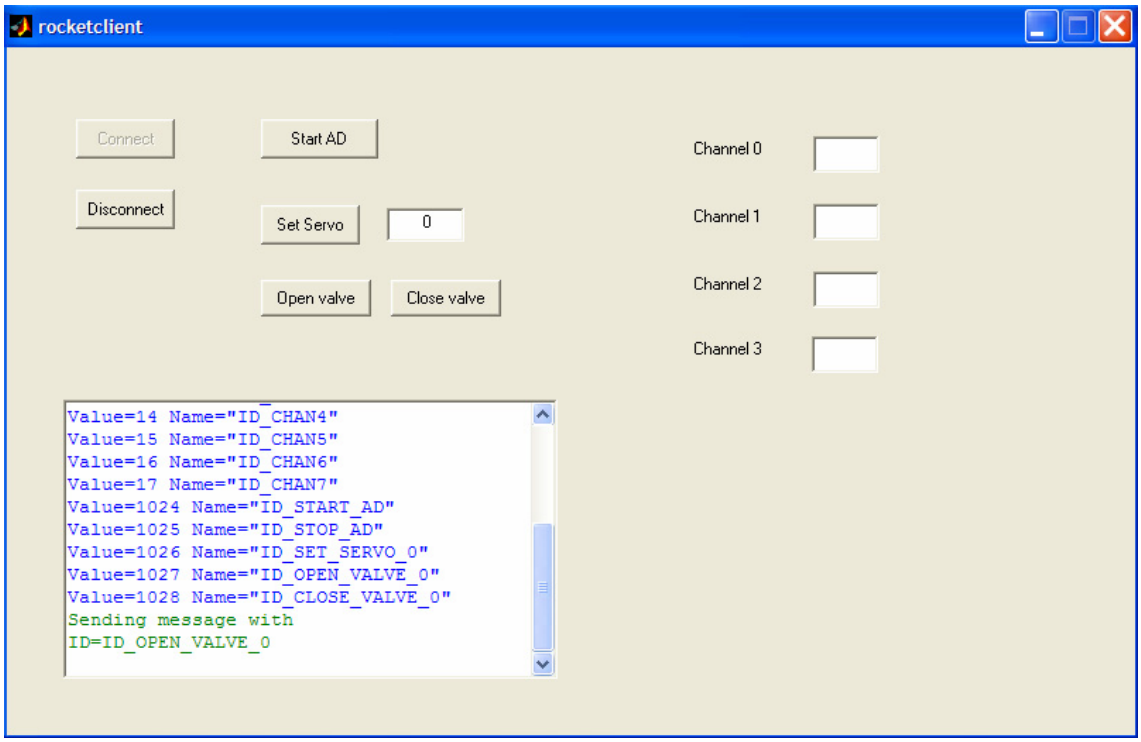


Figure 3.13: MATLAB GUI.

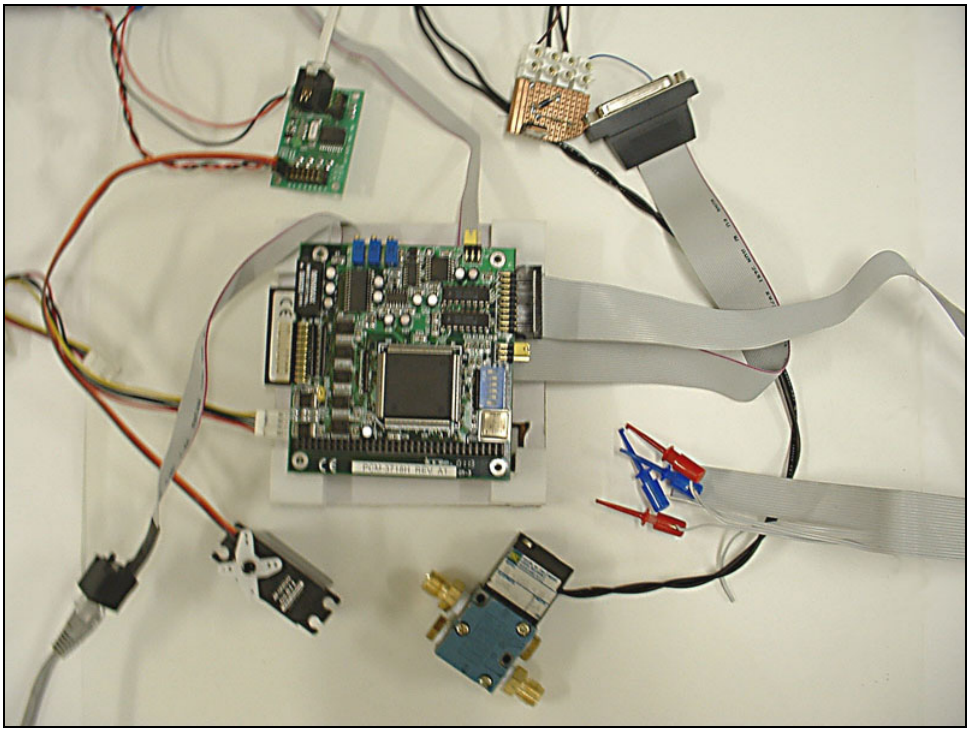


Figure 3.14: The PC/104 controlling servo and solenoid valve. The connectors for data acquisition are located to the mid-right in the picture. The network cable is seen in the lower left corner.

3.7 Chapter summary

In this chapter we have discussed how real-time components can be used for assembling a control program for the hybrid rocket.

Using the QNX operating system offers many advantages, such as real real-time support. We have demonstrated how the operating system in conjunction with software can control servos, solenoid valves and DAQ hardware. A short explanation of how the PC/104 can be monitored and controlled using a remote computer was also presented.

The solutions are vital pieces of the control program architecture. Fitting the pieces together in order to create a single program is the next logical step. This program has been partially implemented but some vital parts are still missing, such as:

- Inertial measurement algorithm
- Autonomous propulsion control (rocket ignition steps)
- Attitude controller
- (Autonomous) guidance controller
- Parachute release control

These parts have to be further analyzed and developed before the rocket control system is ready for testing.

4 Rocket dynamic analysis

In order to control a rocket in flight, it is necessary to make a dynamical analysis of its performance. This includes formulating the equations of motion, accounting for the rigid body motion, the excitation of the aerodynamic forces, rocket engine thrust misalignments and the coupling of the servo system controlling the stability of the rocket attitude [Tho61]. It is evident that a full analysis is beyond the scope of this project. A simplified model may, however, be obtained using Newton's laws in combination with Lagrange's equations. The model can be used as a primary guideline for the formulation of the rocket attitude control system.

4.1 Coordinate system

In the analysis of a moving rocket, it is convenient to use a set of coordinates that is moving along with it. The rocket can be considered to be a system of particles whose position relative the moving coordinate axes can be described by coordinates (x_i,y_i) (see Figure 4.1). The choice of axis orientation is important and depends on the mass distribution of the rocket. In our case we will assume a rocket with uniform mass distribution around its longitudinal axis passing through the centre of mass [Tho61].

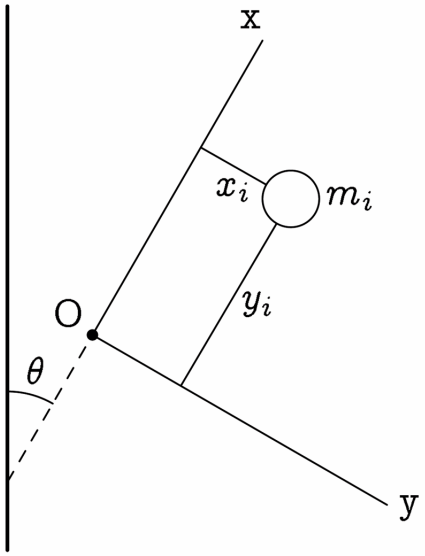


Figure 4.1: The rocket can be considered to be a system of particles.

4.2 Equations of motion for a rocket moving in the plane

We want to acquire a general solution for the rocket motion, i.e. we want to know both its position and its attitude in three-dimensional space. In the most general case this requires us to create a system using six degrees of freedom. A common approach used to describe such systems is to analyze the forces and moments acting on the object.

In order to simplify the transition to a system with six degrees of freedom, we will first consider a rocket moving in a plane. In this case the system will have three degrees of freedom. Choose the *x*-*y* plane as the plane of reference (see Figure 4.2). We assume the rocket to be positioned along the *x*-axis and travelling in the positive *x* direction. For the time being we assume the mass of the rocket to be constant.

The forces in the *x* and *y* directions and the moment about the centre of mass, O, can be deduced directly from Figure 4.2 where *L* and *D* are the lift and drag components of the induced aerodynamic force. F_e is the rocket thrust force and *I* is the moment of inertia. Angular speed $\dot{\theta}$, and angular acceleration $\ddot{\theta}$, are taken as positive in the clockwise direction.

$$\begin{aligned}\sum F_x &= -mg\cos\theta + F_e - D \\ \sum F_y &= mg\sin\theta - L \\ \sum M_O &= I\ddot{\theta}\end{aligned} \tag{4.1a}$$

The forces F_x and F_y can also be written

$$\begin{aligned}\sum F_x &= ma_x \\ \sum F_y &= ma_y\end{aligned} \tag{4.1b}$$

The sum of moments about O may be replaced by a single torque τ acting orthogonally with respect to the longitudinal direction of the rocket (i.e. parallel to the *y*-axis).

$$\sum M_O = \tau \tag{4.1c}$$

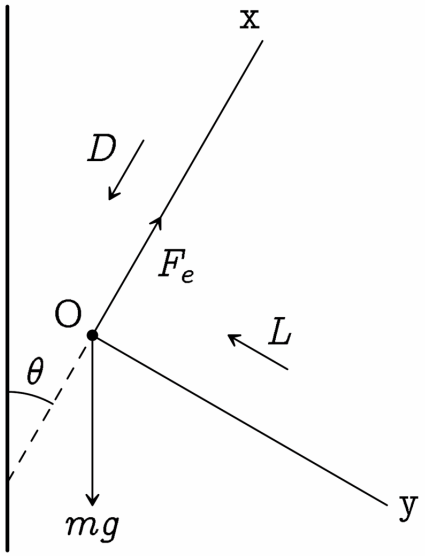


Figure 4.2: Forces acting on the rocket.

Acceleration of the centre of mass O with the *x*- and *y*-components can be described as

$$\begin{aligned}a_x &= \ddot{x}_0 - \dot{y}_0\dot{\theta} \\ a_y &= \ddot{y}_0 + \dot{x}_0\dot{\theta}\end{aligned} \tag{4.2}$$

Combining (4.1) and (4.2) gives the final dynamic formulation of the rocket moving in the plane.

$$\begin{aligned}m(\ddot{x}_0 - \dot{y}_0\dot{\theta}) &= -mg\cos\theta + F_e - D \\ m(\ddot{y}_0 + \dot{x}_0\dot{\theta}) &= mg\sin\theta - L \\ \tau &= I\ddot{\theta}\end{aligned} \tag{4.3}$$

4.3 A simple attitude control system

Consider the system just described. Assume the angle θ to be an attitude angle measured relative some inertial axis parallel to the gravitational force of the Earth (Figure 4.2). By

neglecting the aerodynamical and gravitational forces and assuming the thrust F_e to be constant, a simple rocket attitude control system can be formulated [Dri96].

The part of interest is the last part of equation (4.3) which describes the moment about O. Taking the time dependence into consideration this can be written as

$$\tau(t) = I\ddot{\theta}(t) \tag{4.4}$$

4.3.1 PID controller

We may obtain a PID controller for the system described by (4.4) by performing a Laplace transformation:

$$\mathrm{T}(s) = I\left(s^2\Theta(s) - s\theta(0) - \dot{\theta}(0)\right) \tag{4.5}$$

Assuming $\dot{\theta}(0)$ and $\theta(0)$ to be 0 for convenience, the transformed function $\Theta(s)$ becomes

$$\Theta(s) = \frac{1}{I \cdot s^2}\mathrm{T}(s) \tag{4.6}$$

Adding a PID controller, the system shown in Figure 4.3 is obtained. Variables used are Θ_d, K, T_i and T_d which represents the desired angle and the three parameters of the PID controller.

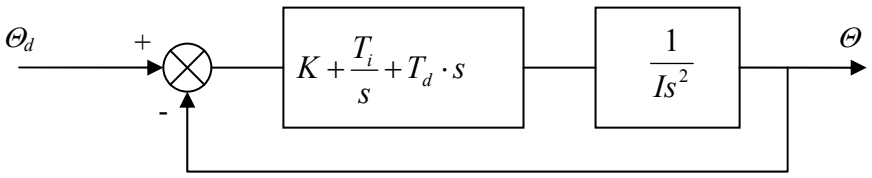


Figure 4.3: PID-controlled system.

The closed-loop form of the system in Figure 4.3 becomes

$$\frac{\Theta(s)}{\Theta_d(s)} = \frac{T_d \cdot s^2 + K \cdot s + T_i}{I \cdot s^3 + T_d s^2 + K \cdot s + T_i} \tag{4.7}$$

4.3.2 State-space controller with integral action

Using (4.4) the system can also be described by two states x_1 and x_2 corresponding to the variables θ and $\dot{\theta}$ respectively. A state-space model may then be obtained as follows:

$$\begin{aligned} \dot{x}_1 &= \dot{\theta} = x_2 \\ \dot{x}_2 &= \ddot{\theta} = \frac{1}{I}\tau = \frac{1}{I}u \end{aligned} \tag{4.8}$$

$$\begin{bmatrix} \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \mathbf{A}x + \mathbf{B}u = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 1/I \end{bmatrix}u \tag{4.9}$$

$$y = \mathbf{C}x + (\mathbf{D}u) = \begin{bmatrix} 1 & 0 \end{bmatrix}\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \tag{4.10}$$

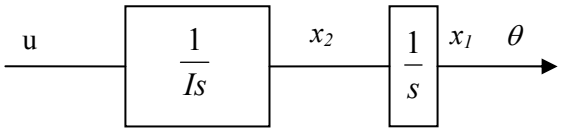


Figure 4.4: State-space model.

The state-space model can be controlled using a linear controller which may be further extended by the addition of integral action. In effect the model now acquires the properties of a PI or PID controller, thus making it more robust [FPE02]. The desired system (Figure 4.5) is obtained by augmenting an extra state x_I into our previous state-space representation.

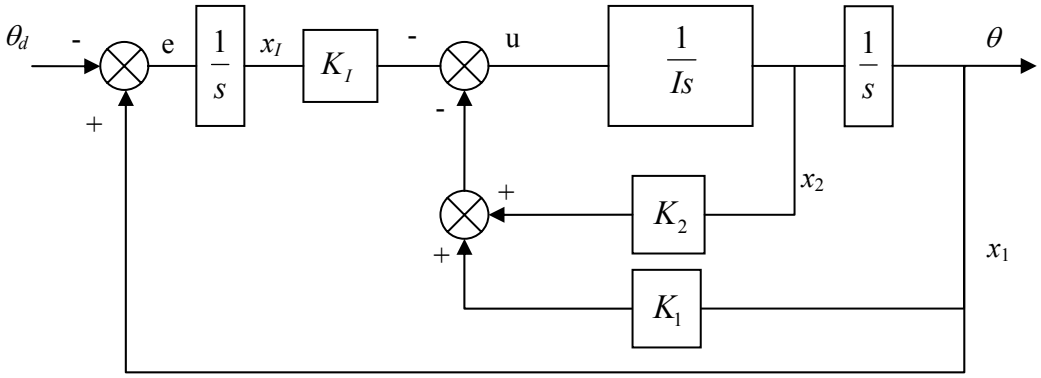


Figure 4.5: State-space model with integration added.

From Figure 4.5 the control error e may be expressed as

$$e = \dot{x}_I = \theta - \theta_d \tag{4.11}$$

The state space representation **A**, **B**, **C** (and **D)** is now augmented with this new state, forming the matrices $\mathbf{A}_\mathrm{I}$, $\mathbf{B}_\mathrm{I}$, $\mathbf{C}_\mathrm{I}$ (and $\mathbf{D}_\mathrm{I}$). This is done by simply adding an extra row (and/or column) containing the new information.

$$\begin{bmatrix} \dot{x}_I \\ \dot{x}_1 \\ \dot{x}_2 \end{bmatrix} = \mathbf{A}_\mathrm{I}\mathbf{x} + \mathbf{B}_\mathrm{I}u - \mathbf{B}_\mathrm{IR}\theta_d = \begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_I \\ x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ 1/I \end{bmatrix} u - \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \theta_d \tag{4.12}$$

$$y = \mathbf{C}_\mathrm{I}\mathbf{x} = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_I \\ x_1 \\ x_2 \end{bmatrix} \tag{4.13}$$

The last part of (4.12) includes the addition of a new matrix, denoted $\mathbf{B}_\mathrm{IR,}$ representing the input from the reference θ_d.

From Figure 4.5, the control law u becomes

$$u = (-\mathbf{K}) \cdot x = -K_I \cdot x_I - K_1 \cdot x_1 - K_2 \cdot x_2 \tag{4.14}$$

Thus, the final state-space model is described by Equations (4.12), (4.13) and (4.14).

4.4 Rocket motion in three dimensions

So far, we have focused on the simplified problem of a rocket moving in two dimensions. We assumed the rocket mass to be constant, and neglected the aerodynamical and gravitational forces. This will not model the rocket motion very accurately. Extending the model to three dimensions is a first step towards a more accurate model.

4.4.1 Equations of motion for a rigid body

An object moving in three dimensions has six degrees of freedom. Three which are translational and three which describe the object's rotation about its centre of mass, all with respect to some axes fixed in the body (known as the body frame). In order to obtain the equations of motion, it is often simpler to start from the kinetic and potential energies of the system and then use Lagrange's equations.

4.4.2 Lagrange's equations for moving coordinates

Euler-Lagrange's formulation of analytical mechanics is fundamental in any advanced approach of problems related to classical mechanics. Using the system's potential and kinetic energies the equations of motion can be derived using Lagrange's equation of motion, which may be stated as

$$\frac{d}{dt}\left(\frac{\partial \mathbf{L}}{\partial \dot{q}_i}\right) - \frac{\partial \mathbf{L}}{\partial q_i} = 0 \tag{4.15}$$

where q_i is a generalized coordinate. **L** is an operator known as the *Lagrangian* defined by

$$\mathbf{L} = \mathbf{T} - \mathbf{V} \tag{4.16}$$

where **T** is the kinetic energy and **V** is the potential energy.

For translational coordinates, Lagrange's equations take a slightly different form, other than the equivalent equations for static coordinates (above) [Tho61]. The exact derivation is lengthy and is left out in favour of a brief explanation (see the book by Thomson [Tho61] for further reference).

The linear momentum **P** can be represented by

$$\mathbf{P} = \frac{\partial \mathbf{T}}{\partial \dot{x}_0} i + \frac{\partial \mathbf{T}}{\partial \dot{y}_0} j + \frac{\partial \mathbf{T}}{\partial \dot{z}_0} k \tag{4.17}$$

Since force has the dimension rate of change of the linear momentum, we have

$$\mathbf{F} = \dot{\mathbf{P}} + \mathbf{\Omega} \times \mathbf{P} \tag{4.18}$$

where $\mathbf{\Omega} = \omega_x i + \omega_y j + \omega_z k$ represents the angular velocities of the body in its body frame. Inserting (4.17) into (4.18) yields the Lagrange equations for the translational movement.

$$\sum F_x = \frac{d}{dt}\frac{\partial \mathbf{T}}{\partial \dot{x}_0} + \omega_y \frac{\partial \mathbf{T}}{\partial \dot{z}_0} - \omega_z \frac{\partial \mathbf{T}}{\partial \dot{y}_0}$$
$$\sum F_y = \frac{d}{dt}\frac{\partial \mathbf{T}}{\partial \dot{y}_0} + \omega_z \frac{\partial \mathbf{T}}{\partial \dot{x}_0} - \omega_x \frac{\partial \mathbf{T}}{\partial \dot{z}_0} \qquad (4.19)$$
$$\sum F_z = \frac{d}{dt}\frac{\partial \mathbf{T}}{\partial \dot{z}_0} + \omega_x \frac{\partial \mathbf{T}}{\partial \dot{y}_0} - \omega_y \frac{\partial \mathbf{T}}{\partial \dot{x}_0}$$

We partially verify (4.19) in the following example:
Assume the kinetic energy is defined by

$$\mathbf{T} = \frac{1}{2} m\left(v_x{}^2 + v_y{}^2\right) \qquad (4.20)$$

i.e. we use a two-dimensional system for simplicity. Note that for a moving coordinate system, the potential energy is zero, i.e. **V**=0; this explains the absence of **V** in (4.19). Now, inserting **T** in (4.17), we obtain for the linear momentum

$$P_x = \frac{\partial \mathbf{T}}{\partial v_x} = m v_x$$
$$P_y = \frac{\partial \mathbf{T}}{\partial v_y} = m v_y \qquad (4.21)$$
$$P_z = 0$$

The linear momentum is defined as mass times velocity. In order to form the cross product between **Ω** and **P** we assume that the frame rotates around the *z*-axis only (i.e. $\mathbf{\Omega} = \omega_z k$).

$$\mathbf{\Omega} \times \mathbf{P} = \begin{bmatrix} -m v_y \omega_z \\ m v_y \omega_z \\ 0 \end{bmatrix} \qquad (4.22)$$

After inserting (4.21) and (4.22) into (4.18) we obtain

$$F_x = m\left(\dot{v}_x - m v_y \omega_z\right)$$
$$F_y = m\left(\dot{v}_y + m v_z \omega_z\right) \qquad (4.23)$$
$$F_z = 0$$

Note that we have obtained the two-dimensional system presented in (4.3). We have confirmed that equation (4.19) applies at least for the two-dimensional case. We will derive the three-dimensional case in section 4.4.4.

4.4.3 Moments of inertia

In order to form the three-dimensional moment equations we will use a similar approach as above (extracted from Fortescue and Stark [FS95]).

The moment about the centre of mass C, $\mathbf{M}_C$ can be represented by

$$\mathbf{M}_C = \dot{\mathbf{H}}_C + \mathbf{\Omega} \times \mathbf{H}_C \qquad (4.24)$$

where $\mathbf{H}_C$ is the angular momentum, which may be expressed using the inertia matrix $\mathbf{I}_C$.

$$\mathbf{H}_C = \mathbf{I}_C \boldsymbol{\Omega} \tag{4.25}$$

$$\mathbf{I}_C = \begin{bmatrix} I_{xx} & I_{xy} & I_{zx} \\ I_{xy} & I_{yy} & I_{yz} \\ I_{zx} & I_{yz} & I_{zz} \end{bmatrix} \tag{4.26}$$

It is obvious that $\mathbf{H}_C$ now is

$$\mathbf{H}_C = \begin{bmatrix} I_{xx}\omega_x + I_{xy}\omega_y + I_{zx}\omega_z \\ I_{xy}\omega_x + I_{yy}\omega_y + I_{yz}\omega_z \\ I_{zx}\omega_x + I_{yz}\omega_y + I_{zz}\omega_z \end{bmatrix} \tag{4.27}$$

Inserting (4.27) into (4.24) and assuming principal axes, the resulting equations for the moments become

$$\begin{aligned} \sum M_x &= \frac{d}{dt}(I_{xx}\omega_x) + (I_{zz} - I_{yy})\omega_y\omega_z \\ \sum M_y &= \frac{d}{dt}(I_{yy}\omega_y) + (I_{xx} - I_{zz})\omega_x\omega_z \\ \sum M_z &= \frac{d}{dt}(I_{zz}\omega_z) + (I_{yy} - I_{xx})\omega_x\omega_z \end{aligned} \tag{4.28}$$

Note that there is a cross-coupling between the three axes. This will pose a problem to the rocket attitude control. Forcing rotations around more than one axis at a time will give rise to additional rotations around all axes.

4.4.4 Equations of motion for a constant mass system

We now derive the equations of motion in the general three-dimensional case. For a rigid body, the kinetic energy is usually represented by

$$\mathbf{T} = \frac{1}{2}m\mathbf{V}^2 + \frac{1}{2}\boldsymbol{\Omega}^T\mathbf{I}_C\boldsymbol{\Omega} \tag{4.29}$$

The derivatives of (4.29) will, however, not contain terms representing the angular velocities. Therefore (4.29) can simply be written

$$\mathbf{T} = \frac{1}{2}m\mathbf{V}^2 = \frac{1}{2}m\left(v_x{}^2 + v_y{}^2 + v_z{}^2\right) \tag{4.30}$$

Inserting (4.30) into (4.19), using (4.28), we obtain (4.31) which fully describes the dynamic motion of a rigid body. In this case, we still assume the mass to be constant.

$$
\begin{aligned}
\sum F_x &= m\left(\dot{v}_x + \omega_y v_z - \omega_z v_y\right) \\
\sum F_y &= m\left(\dot{v}_y + \omega_z v_x - \omega_x v_z\right) \\
\sum F_z &= m\left(\dot{v}_z + \omega_x v_y - \omega_y v_x\right) \\
\sum M_x &= I_{xx}\dot{\omega}_x + (I_{zz} - I_{yy})\omega_y\omega_z \\
\sum M_y &= I_{yy}\dot{\omega}_y + (I_{xx} - I_{zz})\omega_x\omega_z \\
\sum M_z &= I_{zz}\dot{\omega}_z + (I_{yy} - I_{xx})\omega_x\omega_y
\end{aligned}
\tag{4.31}
$$

The equations for **F** agree with the results from Greenwood [Gre65], where a different approach is used.

4.4.5 Equations of motion for a variable mass system

It is fairly obvious that a travelling rocket is not a system that can be assumed to have constant mass, not even approximately. In order to compensate for this we assume that both the kinetic energy as well as the moments of inertia depend on the mass-change of the system. Hence, we need to modify the equations of motion.

We now see the advantage of writing the translational equations using Lagrange's equations. Using the same definitions for the kinetic energy as before, but assuming that the mass is time-dependent we can derive the new equations of motion directly from (4.19). The same applies to the moment equations if we use (4.28), where we have not yet taken the time derivative of $\mathbf{H}_C$ because of this. After applying the product rule of differentiation, it is easy to see that the new system only differs from (4.31) in one term in each equation. System (4.32) now describes the variable mass system.

$$
\begin{aligned}
\sum F_x &= \dot{m}v_x + m\left(\dot{v}_x + \omega_y v_z - \omega_z v_y\right) \\
\sum F_y &= \dot{m}v_y + m\left(\dot{v}_y + \omega_z v_x - \omega_x v_z\right) \\
\sum F_z &= \dot{m}v_z + m\left(\dot{v}_z + \omega_x v_y - \omega_y v_x\right) \\
\sum M_x &= \left(\frac{d}{dt}I_{xx}\right)\omega_x + I_{xx}\dot{\omega}_x + (I_{zz} - I_{yy})\omega_y\omega_z \\
\sum M_y &= \left(\frac{d}{dt}I_{yy}\right)\omega_y + I_{yy}\dot{\omega}_y + (I_{xx} - I_{zz})\omega_x\omega_z \\
\sum M_z &= \left(\frac{d}{dt}I_{zz}\right)\omega_z + I_{zz}\dot{\omega}_z + (I_{yy} - I_{xx})\omega_x\omega_y
\end{aligned}
\tag{4.32}
$$

Bear in mind, that even if we solve for v_x, v_y and v_z we cannot integrate directly in order to obtain the absolute displacement. This is because the coordinate system we use is attached to the rocket (in which only velocities and accelerations have physical meaning). Thus, we must transform the components of (4.32) to an inertial frame before solving for absolute displacement. The same applies to the rotational components. The transformation relations from the body frame to the inertial frame can be calculated using the theory of Euler angles and rotations, which we will discuss next.

4.5 Euler angles

Any rotation may be described using three angles. If the rotations are written in terms of rotation matrices **X**, **Y** and **Z**, then a general rotation **R** can be written as

$$\mathbf{R}=\mathbf{ZYX} \tag{4.33}$$

The three angles needed in the three rotation matrices are known as *Euler angles*. There are several conventions for Euler angles, depending on the axes about which the rotations are defined. A common way of expressing the Euler angles is the "roll-pitch-yaw"-convention used as standard practice for aircrafts. The three angles are *roll* ϕ, *pitch* θ and *yaw* ψ, which defines the rotations about the *x*- *y*- and *z*-axes respectively [FS95]. The rotation is expressed in reverse sequence (*z, y, x*). The respective rotation matrices are

$$\mathbf{Z}=\begin{bmatrix}\cos\psi & \sin\psi & 0\\ -\sin\psi & \cos\psi & 0\\ 0 & 0 & 1\end{bmatrix} \tag{4.34}$$

$$\mathbf{Y}=\begin{bmatrix}\cos\theta & -\sin\theta & 0\\ 0 & 1 & 0\\ \sin\theta & \cos\theta & 0\end{bmatrix} \tag{4.35}$$

$$\mathbf{X}=\begin{bmatrix}1 & 0 & 0\\ 0 & \cos\phi & \sin\phi\\ 0 & -\sin\phi & \cos\phi\end{bmatrix} \tag{4.36}$$

Using standard notation for the elements of a matrix, **R** is defined as

$$\mathbf{R}=\begin{bmatrix}r_{11} & r_{12} & r_{13}\\ r_{21} & r_{22} & r_{23}\\ r_{31} & r_{32} & r_{33}\end{bmatrix} \tag{4.37}$$

we can express the product **R=ZYX** element by element as follows

$$\begin{aligned}
r_{11} &= \cos\theta\cos\psi\\
r_{21} &= \cos\theta\sin\psi\\
r_{31} &= -\sin\theta\\
r_{12} &= \sin\phi\sin\theta\cos\psi - \cos\phi\sin\psi\\
r_{22} &= \sin\psi\sin\theta\sin\phi + \cos\psi\cos\phi\\
r_{32} &= \cos\theta\sin\phi\\
r_{13} &= \cos\phi\sin\theta\cos\psi + \sin\phi\sin\psi\\
r_{23} &= \cos\phi\sin\theta\sin\psi - \sin\phi\cos\psi\\
r_{33} &= \cos\theta\cos\phi
\end{aligned} \tag{4.38}$$

4.5.1 Choice of coordinate set

Resorting to aircraft standard for the choice of reference directions, we should place the *x*-axis in the nominal direction of travel (roll-axis), the *z*-axis nominally down towards the local vertical (yaw-axis) and the *y*-axis such that it completes the right-hand system (pitch-axis).

In the case of a rocket it is worth noting a potential problem using this choice of axes. If the pitch angle θ is 90° then there is a singularity in the Euler angles. If the rocket is launched vertically its pitch attitude would be 90° and its yaw and roll angles cannot be uniquely resolved. Such problems can be overcome with a more suitable choice of axes [FS95].

4.5.2 Resolving the Euler angles in flight

Using the Euler angles and rotations described, we may deduce the relationship between the Euler angles and the angular velocities of the aircraft. The result can be found in [FS95] and is reproduced below.

$$\begin{aligned} \omega_x &= \dot{\phi} - \dot{\psi}\sin\theta \\ \omega_y &= \dot{\theta}\cos\phi + \dot{\psi}\cos\theta\sin\phi \\ \omega_z &= \dot{\psi}\cos\theta\cos\phi - \dot{\theta}\sin\phi \end{aligned} \tag{4.39}$$

More important is the inverse relationship

$$\begin{aligned} \dot{\psi} &= (\omega_y \sin\phi + \omega_z \cos\phi)/\cos\theta \\ \dot{\theta} &= \omega_y \cos\phi - \omega_z \sin\phi \\ \dot{\phi} &= \omega_x + (\omega_y \sin\phi + \omega_z \cos\phi)\tan\theta \end{aligned} \tag{4.40}$$

For small angles, then $\dot{\psi} \approx \omega_z$, $\dot{\theta} \approx \omega_y$ and $\dot{\phi} \approx \omega_x$.

Integrating (4.40) the attitude in form of the Euler angles can be obtained from measured components of the angular velocities. Note that the singularity at θ=90° appears in the form of a tan θ which leads to problems performing the integration [FS95].

The matrix for transforming the velocity vector from the axes fixed to the aircraft $\mathbf{V_{a/c}}$ to the inertial axes $\mathbf{V_{Inertial}}$ is

$$\mathbf{V_{Inertial}} = \mathbf{ZYX}\ \mathbf{V_{a/c}} \tag{4.41}$$

By solving for v_x, v_y and v_z in the inertial frame and integrating, the absolute displacement of the rocket may be obtained.

4.5.3 Relationship with the IMU

The inertial measurement unit uses the mathematics described in the section above for deducing the attitude and position relative to the inertial axes. Three accelerometers measure acceleration in each of the principal directions of the aircraft and three gyros monitor ω_x, ω_y and ω_z. The transformation and integration concerning the position as well as the attitudes should be performed by the inertial measurement algorithm mentioned in section 3.3.1. Previous research and algorithm design have been performed for the IMU system.

4.6 Dynamic modelling using MATLAB Simulink

Let us now return to section 4.4, where we formulated the equations of motion for a system of non-constant mass with six degrees of freedom (Equation (4.32)). In this section we concentrate on the equations for the moments of inertia, aiming at a model of the dynamical behaviour of the rocket attitude.

In order to better understand the dynamics, we need to perform computer simulations and analysis using numerical software. For this purpose the MATLAB software suite was used. It incorporates a dynamic modelling environment as well as sophisticated tools necessary for this type of analysis. The modelling environment is called Simulink. It provides a graphical interface where systems are built using blocks and connections. With none or little modifications we implement our equations as Simulink blocks.

It is advantageous to represent our system in state-space form since this is more suitable for computer implementation. The states used are

$$\begin{aligned} x_1 &= \theta_x & \mathrm{x}_2 &= \dot{\theta}_x = \omega_x \\ y_1 &= \theta_y & \mathrm{y}_2 &= \dot{\theta}_y = \omega_y \\ z_1 &= \theta_z & \mathrm{z}_2 &= \dot{\theta}_z = \omega_z \end{aligned} \tag{4.42}$$

The input signals are the general moments of inertia (M_x, M_y and M_z) which in some sense represent the physical relationship with the moving fins placed on the rocket body and the torques they induce on the body itself. Unfortunately, we are not able to describe this relationship using an analytical formulation. Instead we have to rely on wind tunnel tests with the rocket once it has been assembled, or finite element simulations. When this relationship is better understood it is possible to add it to the model. For now, we leave these inputs as is and simply refer to them as input signals U_x, U_y and U_z, respectively.

Using (4.32) and (4.42) we can formulate the state-space representation.

$$\begin{aligned} \dot{x}_1 &= x_2 & \dot{\mathrm{x}}_2 &= \frac{1}{I_{xx}}\left(U_x - \left(\frac{d}{dt}I_{xx}\right)x_2 - \left(I_{zz} - I_{yy}\right)y_2 z_z\right) \\ \dot{y}_1 &= y_2 & \dot{\mathrm{y}}_2 &= \frac{1}{I_{yy}}\left(U_y - \left(\frac{d}{dt}I_{yy}\right)y_2 - \left(I_{xx} - I_{zz}\right)x_2 z_2\right) \\ \dot{z}_1 &= z_2 & \dot{\mathrm{z}}_2 &= \frac{1}{I_{zz}}\left(U_z - \left(\frac{d}{dt}I_{zz}\right)z_2 - \left(I_{yy} - I_{xx}\right)x_2 y_2\right) \end{aligned} \tag{4.43}$$

Again, the x, y and z-axes are the roll, pitch and yaw-axes respectively. Also, we assume the moments of inertia to lie along the principal axes and that they form an orthogonal system with the x axis, i.e. $I_{yy} = I_{zz}$.

4.6.1 Moments of inertia estimator

A simple method for estimating the moments of inertia is to assume the mass distribution to be homogenous in some simple geometric shape. For a rocket, a solid cylinder or a solid cone might be a good choice. Although the rocket does not resemble a cone by shape, its mass distribution can be assumed to have a different form than its physical shape. This is especially useful when the cylindrical objects do not have a symmetrical distribution of mass. This technique is often used in robotics [McK91]. Considering the density of the rocket motor and its propellant, it is probable that the centre of mass will lie closer to the motor itself rather than at the geometrical centre of the rocket. For this reason, we will consider both the solid

cylinder and the solid cone as possible descriptions of the mass distribution. From tables we find the moments of inertia around the centre of mass for a cylinder (cone) with mass m, radius r and length l.

Solid Cylinder:

$$I_{xx} = \frac{mr^2}{2}$$
$$I_{yy} = I_{zz} = \frac{mr^2}{4} + \frac{ml^2}{12} \tag{4.44}$$

Centre of mass: $l/2$

Solid Cone:

$$I_{xx} = \frac{3}{10}mr^2$$
$$I_{yy} = I_{zz} = \frac{3}{20}mr^2 + \frac{3}{80}ml^2 \tag{4.45}$$

Centre of mass: $l/4$

Using these equations we can create two different block components in Simulink (see Figure 4.6). We let the mass m be an input signal to the block/subsystem, representing the current mass of the system. In order to include cross-coupling in the model, we also pre-calculate the two terms $I_{yy} - I_{xx}$ and $I_{xx} - I_{zz}$ (see (4.43)).

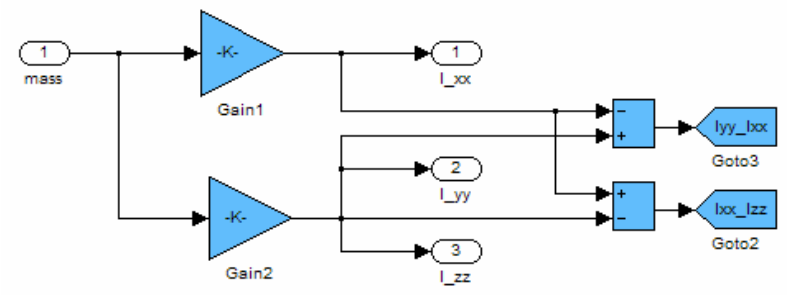


Figure 4.6: Inertia estimator.

The gains found in Figure 4.6 are set to:

$$Gain1 = \begin{cases} \frac{3}{10}r^2 & \text{for the cone} \\ \frac{1}{2}r^2 & \text{for the cylinder} \end{cases}$$

$$Gain2 = \begin{cases} \frac{3}{20}r^2 + \frac{3}{80}l^2 & \text{for the cone} \\ \frac{1}{4}r^2 + \frac{1}{12}l^2 & \text{for the cylinder} \end{cases}$$

The output from this block is the moment of inertia based on the current mass. This may be used by the rest of the system.

4.6.2 Angular velocity estimator

Next, we study the Simulink blocks calculating the states x_2, y_2 and z_2 (or ω_x, ω_y and ω_z). For x_2, there are no cross-coupled terms since we assumed that I_{yy} and I_{zz} are equal. The time derivative of x_2 from (4.43) can thus be stated as

$$\dot{x}_2 = \frac{1}{I_{xx}}\left(U_x - x_2 \frac{d}{dt} I_{xx}\right) \tag{4.46}$$

This can be directly implemented in Simulink block form.

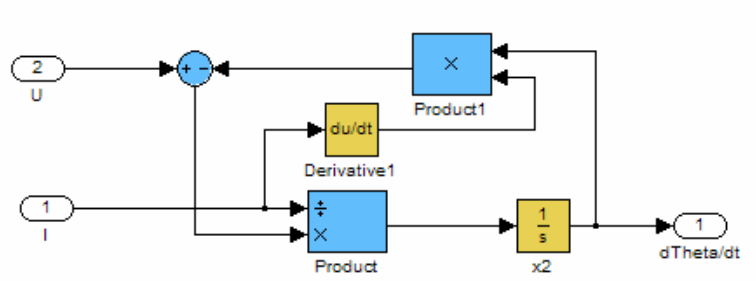


Figure 4.7: Angular velocity estimator without cross-coupling.

U and I are both external signals. U is the torque acting on the axis. I is the corresponding moment of inertia. The output is obtained after the integrator with output x_2.

For the y- and z-axes, the angular velocities depend on the cross-coupled terms. These are included in the respective estimators. An overview of the block calculating y_2 is shown in Figure 4.8. In order to reduce the number of crossed wires, the model uses Goto and From blocks, allowing information to be transferred between the estimator subsystems.

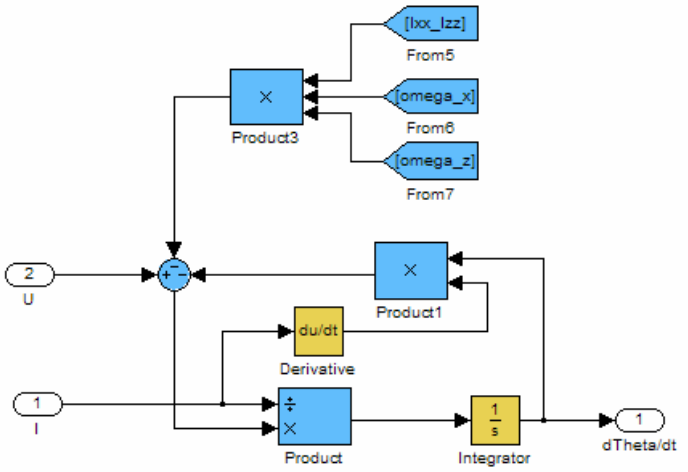


Figure 4.8: Angular velocity estimator with cross-coupling.

4.6.3 Assembly of model

We are now able to assemble our physical model. As input torques U we use constant signals (step functions). The rocket mass is modelled by a ramp function with a negative slope. The full model may be found in Figure 4.9. Four Simulink scopes are employed in the model, as shown in Figure 4.9, recording the rocket mass m, and the three angular velocities ω_x, ω_y and ω_z.

It is not possible to integrate ω_x, ω_y or ω_z in order to obtain the angles of rotation since we are using a moving coordinate system. The angular velocities are needed as inputs for an Euler angle rotation algorithm, based on (4.40). Performing these transformations makes it possible to deduce the attitude angles relative the inertial frame. The angles may in turn be used as inputs to an attitude control system.

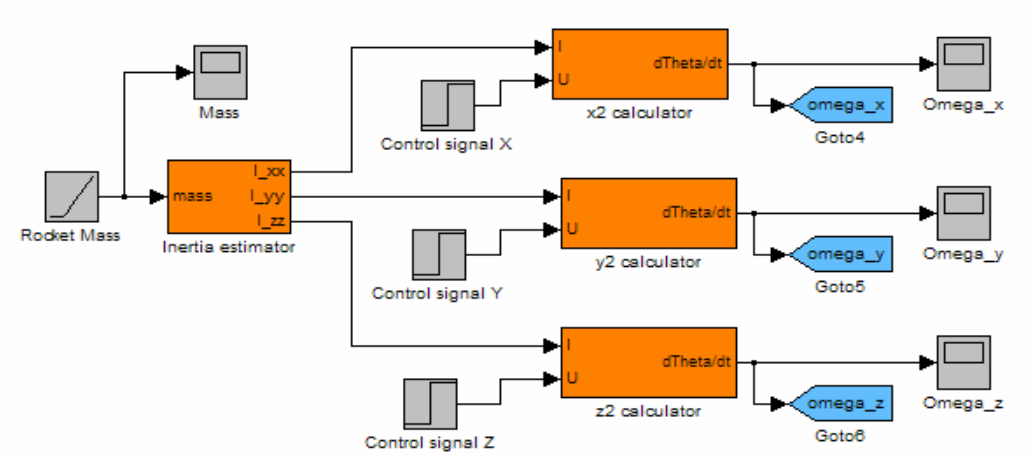


Figure 4.9: Overview of Simulink model.

4.6.4 Model simulations and results

Using the above Simulink model, we can study the effects of different configurations such as initial parameters and input signals. In order to keep the discussion short we use the same starting parameters for all our simulations. These are shown below.

Table 4-1 Simulation parameters.

Parameter	Description	Value
m_0	Total starting mass of the rocket	12 kg
dm/dt	Fuel ejection rate (slope of ramp function)	-160 g/s
L	Length of rocket	1 m
R	Outer radius of rocket	80 mm

(The parameters chosen have physical relevance for the rocket and motor studied in this project)

We only consider the case of a conic mass distribution. The cylindrical distribution shows almost identical behaviour. As a reference, we use the case where there are no cross-coupling between the axes. This can be achieved by letting the rotation around the *x*-axis (roll) be zero. We may then observe e.g. the *y*-axis applying a torque of say, 0.5 Nm. The result is shown in the graph found in Figure 4.10, below.

Since in this case, most of the terms are zero, the angular velocity increases almost linearly. The reason for the increasing slope is that the rocket is loosing mass (making it more sensitive to external disturbances, such as torques).

Next, we add a very small rolling torque around the *x*-axis of say, 0.01 Nm. We still apply the 0.5 Nm torque around the *y*-axis. The result is seen in Figure 4.11. The top graph shows ω_x and the lower ω_y and ω_z. Since we assume that I_{yy} and I_{zz} are equal, no cross-coupling occurs for the *x*-axis, making it look much like the case we had in Figure 4.10. For the *y*- and *z*-axes, however, the result is different.

Both angular velocities start to oscillate, which can be interpreted as if the rocket starts to wobble. This effect is a known source of instability in rocket and aerospace sciences. The solution is often solved by performing only one rotation (manoeuvre) at a time, minimizing the cross-coupling effects.

If we apply torques to all three axes at the same time, the results are similar to the one seen in Figure 4.11. The conclusion from these simulations is that rolling should be avoided while performing yaw or pitch manoeuvres. During actual rocket flight there should be no need for performing roll manoeuvres.

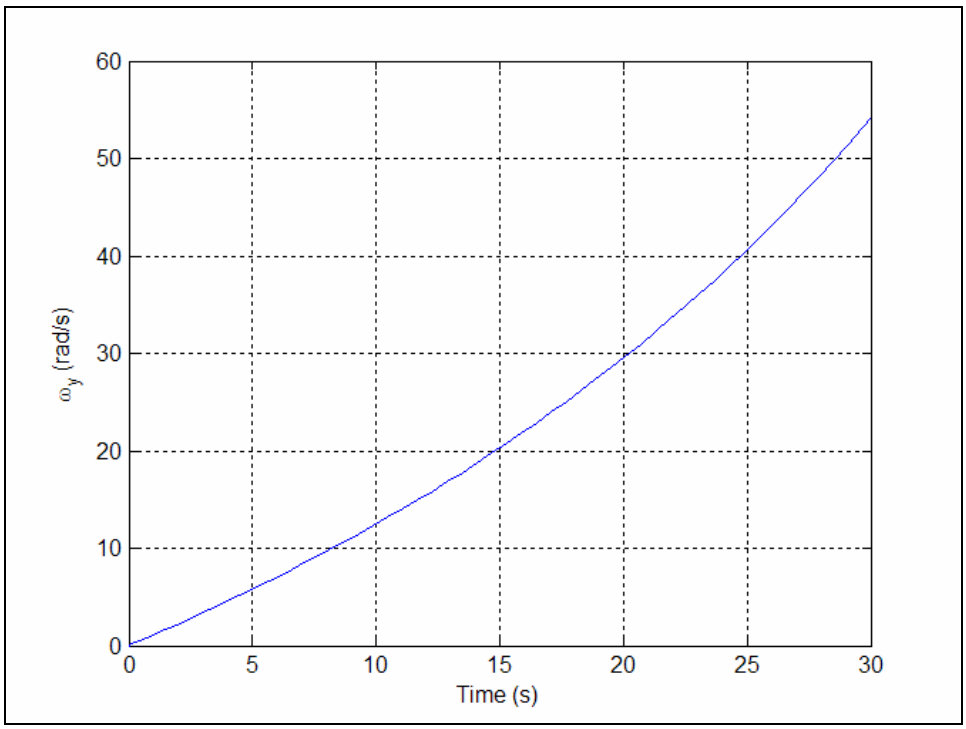


Figure 4.10: Angular velocity for the *y*-axis, with U_y=0.5 Nm and U_x=U_z=0 Nm.

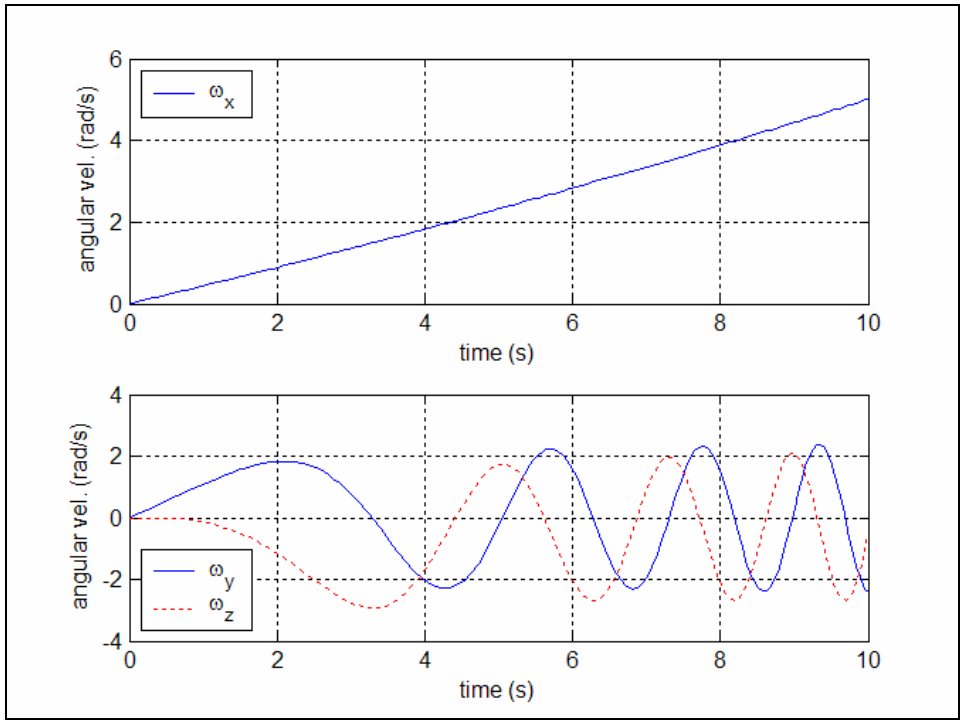


Figure 4.11: Effects of two input signals; U_x=0.01 Nm, U_y=0.5 Nm, U_z=0 Nm.

4.7 Attitude control system

In the previous section we presented the modelling of the physical properties of the rocket dynamics using Simulink. In this final section we will study the performance of a PID controller using the model. The results presented here can with advantage be reused in the implementation phase for the control software in the rocket.

First, we require a system for making the transformation between the aircraft body frame and the inertial frame. Using (4.40) it is straightforward to implement such subsystem in Simulink. The resulting subsystem is shown in Figure 4.12. The internal structure of the

block is cluttered and is not shown. We use conversion from radians to degrees on all Simulink scopes recording angles. Next, we use two PID controllers for replacing the torque generators for the *y*- and *z*-axes in our original model shown in Figure 4.9. The internal structure of the controllers is shown in Figure 4.14. We also replace the names of the control signals *x*, *y*, *z* to *Roll*, *Pitch* and *Yaw*, respectively. A number of new Simulink scopes have also been incorporated into the model (see Figure 4.13).

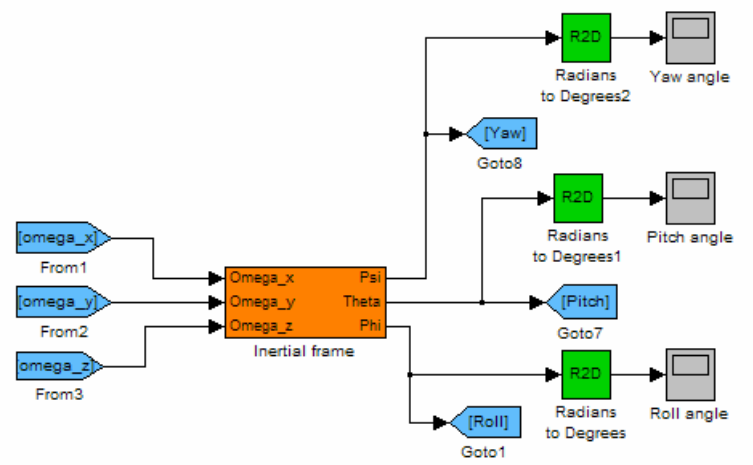


Figure 4.12: Transformation to inertial frame.

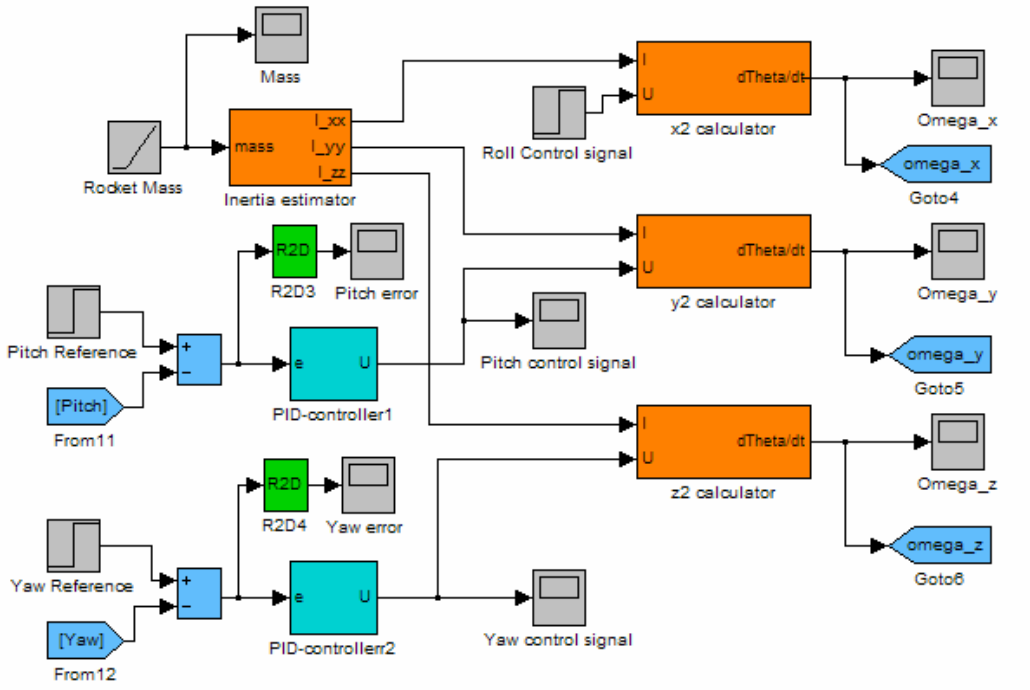


Figure 4.13: PID-controlled model with pitch and yaw controllers.

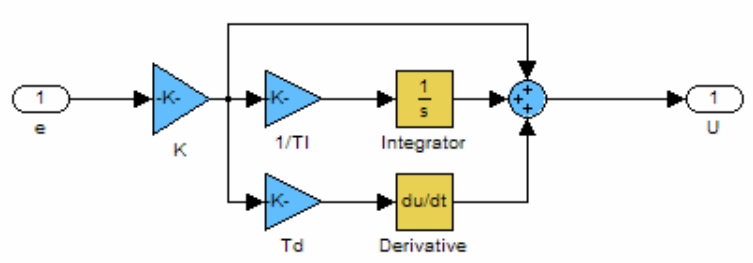


Figure 4.14: Internal structure of PID controller.

We now want to find out if the system shown above in Figure 4.13 is controllable, i.e. if it is at all possible to have separate controllers for each axis, even though the system properties are clearly interconnected through cross-coupling (see (4.32)).

In our first test we neglect the cross-coupled effects of the angular velocities as in section 4.6.4. We thus set the control signal for the *x*-axis to zero. Then by tuning the PID parameters we can analyze the step response of the yaw and pitch controllers. Using the notation of Figure 4.14, the PID parameters is set to K = 13, TI=0.7 and Td=0.2. These parameters have been deduced through repeated simulations and produce a satisfactory step response, which in turn may be found in Figure 4.15.

Examining Figure 4.15 we see that the cross-coupling is almost non-existent. Both the yaw and pitch curves lie on top of each other. One can also notice a small roll effect due to the manoeuvres made.

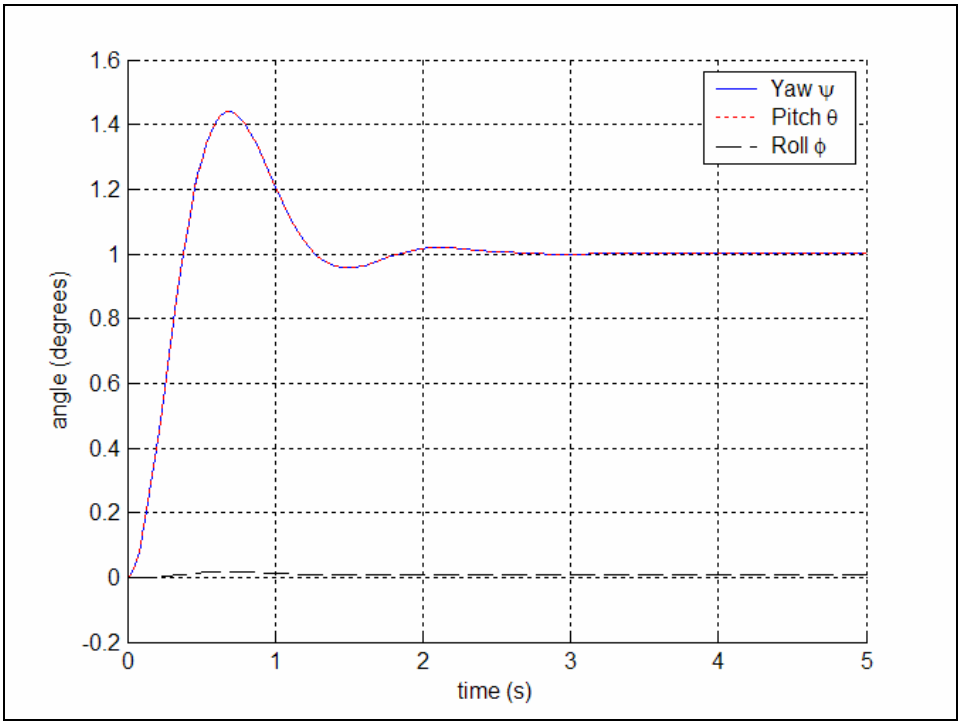


Figure 4.15: Model simulation with $\psi_{ref} = \theta_{ref} = 1°$, $U_X = 0$.

In order to obtain a better idea of how the controller(s) would work in a more realistic situation we also perform a stress test of the system. Starting at yaw, pitch and roll at zero degrees, we forced the system into some new position, say pitch=45° and yaw=60°. We also investigated the effect of a constant torque of typically 10^{-3} Nm around the *x*-axis, thereby causing roll effects as well (acting as a disturbance). As we can see in Figure 4.16 the results appear to be quite promising. Even though the rocket rolls some 45 degrees neither the yaw nor pitch control signals are affected. The overshoot, however, for both controlled signals increases significantly. This is due to the fact that both controllers are performing their manoeuvres simultaneously.

Increasing the rolling torque around the *x*-axis further, causes the system to become unstable after some time of apparent stability. This is probably due to numerical instability of the solution algorithm for some specific part of the model. Unfortunately the analysis of this effect was not conclusive. Nevertheless we have shown that control of the system, which at first seemed very complicated, is indeed possible.

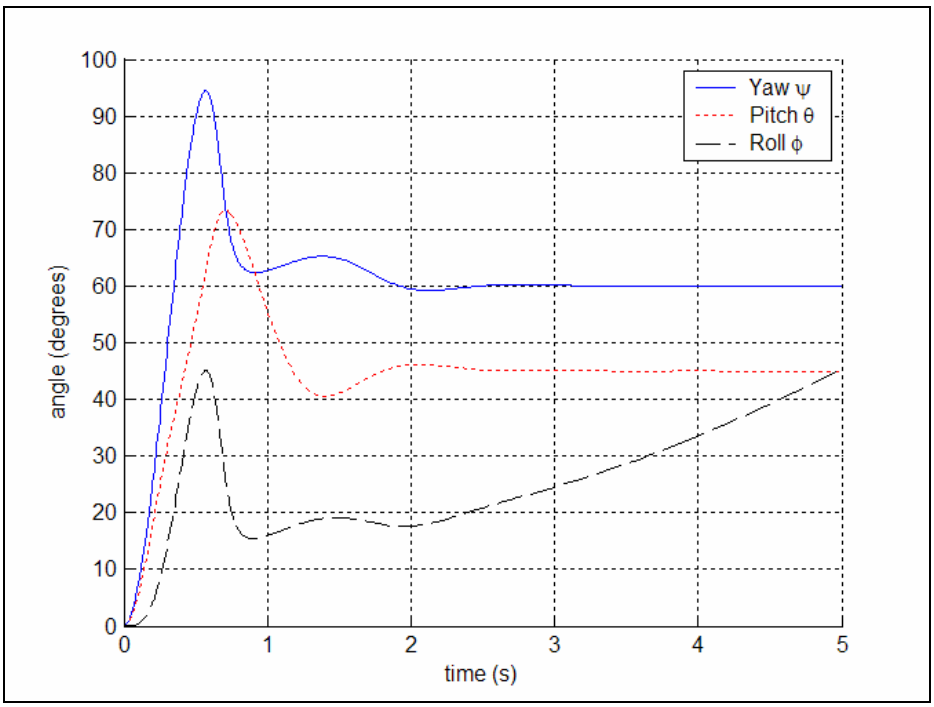


Figure 4.16: Double PID Model simulation with ψ_{ref}= 60°, θ_{ref}= 45°, U_X=10^{-3} Nm.

Finally, we add a PID controller for the *x*-axis. We now have one controller for each of the manoeuvres roll, pitch and yaw. The resulting model is shown in Figure 4.17. The controller parameters are still the same as before (K = 13, TI=0.7, Td=0.2).

We also added disturbance torques on all three axes, acting as generalized forces not accounted for by the model. As disturbance signals we choose sine waves with amplitude 0.1 Nm and frequency 5 rad/s. This kind of disturbances gives more information about the robustness of the controller due to the fact that the system becomes excited by both negative and positive disturbances in the same simulation.

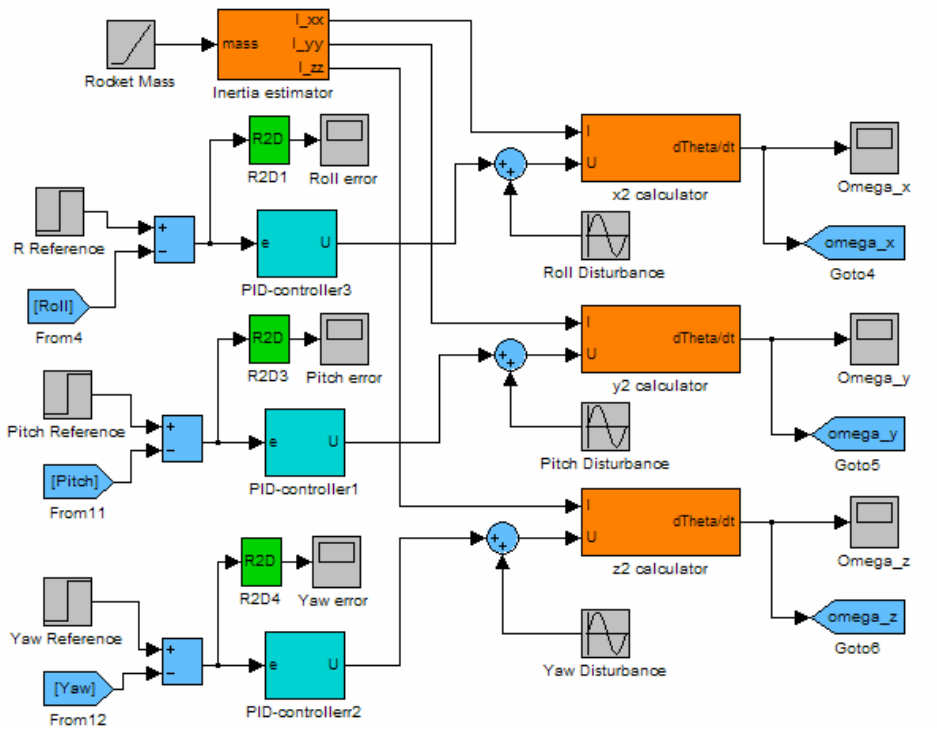


Figure 4.17: Model with roll, pitch and yaw controllers.

The results from the simulation are found in Figure 4.18, where we made the rocket go to pitch=45°, yaw=60° and roll=20°. We see that the PID controllers have no problems reaching their final states. The sine-shaped disturbances are seen as small oscillations, causing effects which the controllers reduce. This shows that separate PID controllers, one for each Euler rotation can be used despite the cross-coupling between the axes.

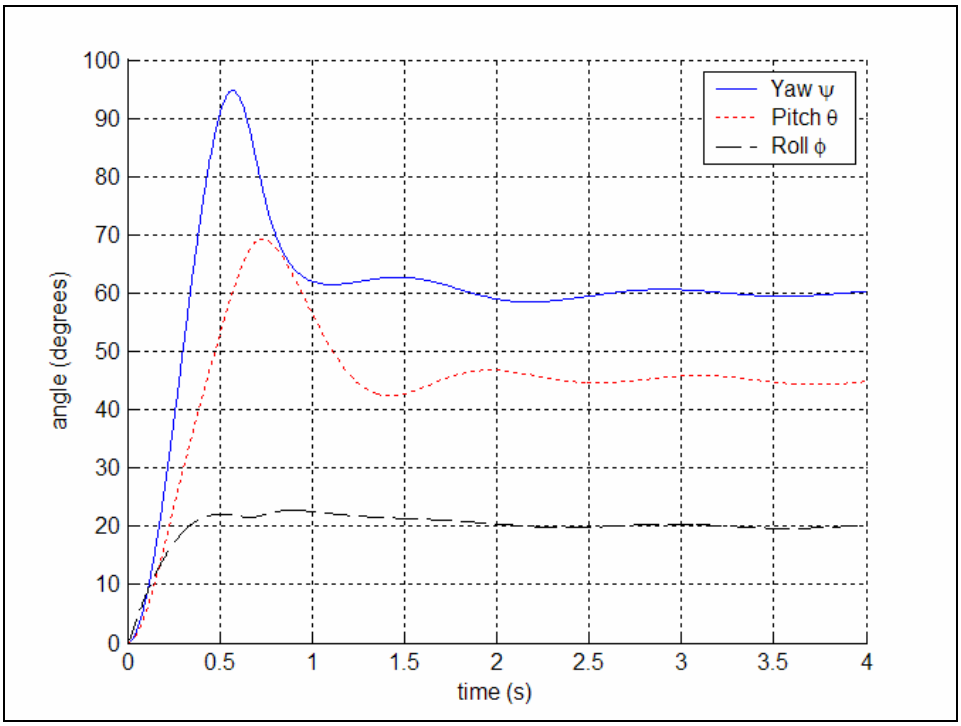


Figure 4.18: Triple PID Model simulation with ψ_{ref}= 60°, θ_{ref}= 45°, ϕ_{ref}=20°.

4.8 Chapter summary

In this chapter we have seen how a physical model for rocket motion can be formulated. Further we have implemented the derived model in MATLAB Simulink, and we have shown how the model can be controlled using separate PID controllers. We have also shown how the attitude angles may be calculated from internal measurements using Euler rotations.

We have further verified that interference from cross-coupled terms do not rule out the employment of separate controllers for each axis. In order to be on the safe side, though, only one manoeuvre should be performed at a time at least in the first tests of the attitude control system (for reducing overshoot).

Please note that the derived model describes a rocket moving in free space, i.e. there are no external forces acting on the system such as gravity or wind. During computer simulations, the influences from external forces were represented as disturbances only. The work and the results presented here are intended as a basis for further study and implementation of the attitude control system.

5 Summary and conclusions

In this thesis we have discussed the aspects of controlling a hybrid rocket. We have suggested an implementation using real-time components, which form the basis for the control software running the onboard PC/104 computer.

We have demonstrated:

- How to open/close solenoid valves, showing in principle how the hybrid rocket engine can be ignited from the software.
- How to perform data acquisition using the developed software driver. This can be used for various purposes, such as inertial measurement or system monitoring.
- How to control servos, showing in principle how the attitude controller can manoeuvre the rocket using movable fins connected to the servos.
- How to collect data and monitor system performance of the hybrid rocket using network communication and data logging.

Further, we have formulated a first model of the rocket dynamics. Using this model we have shown how PID controllers can be used for controlling the rocket attitude.

5.1 Future work

Using the methods and implementation presented, it should be possible to design a complete control system for the onboard PC/104 computer.

The parts and functions not implemented are:

- Inertial measurement
- Autonomous propulsion control (ignition steps)
- Attitude and guidance controller in C code
- Parachute release control

5.1.1 Inertial measurement

The inertial measurement unit needs to be completed and tested. Also the output from the software algorithm that calculates the positions, velocities and rotations need to be connected as input to the attitude controller. If the inertial measurement algorithm is a separate thread in the control program, shared memory areas with protections such as R/W locks (discussed in Chapter 3) or similar may be employed.

5.1.2 Propulsion control

Using the propulsion control interface, described in Chapter 3 (code implementation found in Appendix G) it should be possible to develop a script that performs the necessary steps of the ignition process (see section 3.4.3).

5.1.3 Attitude and guidance controller

Chapter 4 gives an introductory explanation of the rocket dynamics. It does not, however, discuss the relationship between the torques acting on the rocket and the corresponding angles of the movable fins relative the rocket body. This study is required in order for the attitude controller to function. This kind of study is not well suited for a theoretical approach, and we

will have to rely on wind tunnel tests or other types of measurements performed once the rocket has been assembled.

5.1.4 Parachute release control

The parachute release has not yet been studied. Questions such as “Do we need to remove the rocket nose cone before parachute release?” will have to be answered. Both hardware and corresponding software implementations need to be developed.

5.2 Conclusions

Once the issues above have been solved, the hybrid rocket targeted in this report will have been brought closer to its first successful operation. This work constitutes a substantial effort toward that goal, and we wish those who will be continuing the work Good Luck.

References

[Adv02] Advantech Co. Ltd.
PCM-3350 Series NS Geode 586-Level PC/104 CPU Module with VGA/LCD/LAN Interface – User's Manual; 3rd. ed.
Taiwan; Advantech Co. Ltd., 2002
Part No. 2006335052

[Adv99] Advantech Co. Ltd.
PCM-3718H/3718HG; PC/104 12-bit DAS Module with Programmable Gain User's manual; 2nd ed.
Taiwan; Advantech Co. Ltd., 1999
Part No. 2003371810

[BW97] Alan Burns, Andy Wellings
Real-Time Systems and Programming Languages; 2nd ed.
Harlow; Addison-Wesley, 1997

[Dri96] Morris Driels
Linear Control Systems Engineering
New York; McGraw-Hill, 1996

[FPE02] Gene F. Franklin, J. David Powell, Abbas Emami-Naeini
Feedback control of dynamic systems; 4th ed.
Upper Saddle River, N.J.; London: Prentice Hall PTR, 2002

[FS95] Peter W. Fortescue, John P. W. Stark
Spacecraft systems engineering; 2nd ed.
Baffins Lane Chichester; John Wiley & Sons, 1995

[Gre65] Donald T. Greenwood
Principles of dynamics
New York; Prentice-Hall, 1965

[HP91] T. Hayles, D. Potter
Programming DMA on PC/XT/AT Computers
USA; National Instruments Corporation, 1991
See: http://www.inf.pucrs.br/~eduardob/disciplinas/arqii/dma.html

[Krt98] Rob Krten
Getting started with QNX 4: a guide for realtime programmers
Kanata, Ont.; PARSE Software Devices, 1998

[Krt99] Rob Krten
Getting started with QNX Neutrino 2: a guide for realtime programmers
Kanata, Ont.; PARSE Software Devices, 1999

[Lew91] Donald A. Lewine.
POSIX programmer's guide: writing portable UNIX programs with the POSIX.1 standard.
Sebastopol, CA; O'Reilly & Associates, 1991 (Revised 1994)

[MATLAB] Homepage of MATLAB online documentation
Natick, MA; Mathworks Inc., http://www.mathworks.com/support/

[McK91] Phillip John McKerrow
Introduction to robotics
Sydney, Reading, Mass.; Addison-Wesley Pub. Co., 1991

[MSDN] Homepage of Microsoft Developers Network (MSDN)
Redmond, WA; Microsoft Corporation, http://msdn.microsoft.com

[PC104] Homepage of PC/104 Consortium
San Francisco, CA; PC/104 Consortium, http://www.pc104.org

[QNX] Homepage of QNX online documentation and support
Ottawa, Ont.; QNX Software Solutions Ltd, http://www.qnx.com/support/

[Sch98] Herbert Schildt
C++: the complete reference; 3rd ed.
Berkeley; Osborne McGraw-Hill, 1998

[Sch00] Herbert Schildt
Windows 2000 Programming from the ground up
Berkley; Osborne McGraw-Hill, 2000

[Ste98] W. Richard Stevens
UNIX network programming; 2nd ed.
Upper Saddle River, NJ; Prentice Hall PTR, (1998-1999)

[Swe99] Michael R. Sweet
Serial Programming Guide for POSIX Operating Systems 5th ed.
Online documentation, 1999, http://www.easysw.com/~mike/serial/

[Tho61] William Tyrell Thomson
Introduction to Space Dynamics
New York; John Wiley & Sons Inc., 1961

[Tro00] Andrew W. Troelsen
Developer's workshop to COM and ATL 3.0
Plano, TX; Wordware Publishing, 2000

A Analogue to digital conversion

In order to measure and analyze analogue electrical signals in a computer, an analogue to digital converter (ADC) is required. During operation, the analogue signal is measured for a very short time period, usually a few microseconds. This is known as *sampling the signal* and the acquired measurement is called a *sample*. Sampling gives an accurate measurement at the particular time-instant where the sample is made.

By sampling often enough it is possible to fully reconstruct the continuous version of the signal using only the samples. This is a direct consequence of the fundamental theorem of sampling. There is an important relationship between the sampling frequency f_s and the maximum frequency of the signal f_M:

$$f_s \geq 2 f_M$$

The frequency $f_s=2f_M$ is known as the *Nyqvist frequency* and is equivalent to the maximum frequency that can be detected or reconstructed. This frequency becomes very important when dimensioning measurement systems. The question "How often is often enough" is always reoccurring when working with analogue measurement systems.

The frequency f_M is the maximum frequency allowed as input to the measurement system. If there is indication that the signal could contain higher frequencies than f_M then the signal need to be filtered before it enters the ADC. Otherwise aliasing will distort the converted signal. In other words, a low-pass filter should always accommodate an analogue to digital measurement solution.

A.1 Range, accuracy and resolution

An ADC, like any measurement device is classified by three characteristics:

- **Range** – The input and output range where the device operates.
- **Accuracy** – The statistical accuracy of the measurement.
- **Resolution** – The sensitivity of the device (change of output divided by change of input).

It is possible to freely choose in which range the converter operates. The choice of range widely affects the resolution of the measurement as will be shown. The accuracy however is often dependent on the hardware itself. For this purpose potentiometers for tuning are often located in conjunction with the ADC.

A.2 Digital representation

Sampled signals are stored in a computer using digital representation, i.e. bits. The more bits used for acquiring a sample, the higher resolution is obtained. For example if eight bit resolution is used, then $2^8=256$ unique values can be used to represent the measurement. In a computer, digital words start at zero. This means that all words range from 0 to 2^n-1 where n is the number of bits used for the storage. For example, the range would become 0-255 if eight bits are being used.

The most common type of analogue converter cannot fully represent the entire measurement range specified by the manufacturer (this is an inherent property of the converter). We will show this by an example. Suppose we have an 8-bit ADC with input range 0-5 Volts. We then have a maximum of 256 unique values for representing the entire interval. By dividing the reference voltage by the number of unique values, we obtain the resolution.

$$resolution = \frac{V_{ref}}{2^n}$$

For a 5V reference, the resolution is:
5V/256 = 0.0195V or 19.5mV

The 8-bit converter represents the analogue input as a digital word. The most significant bit of the word indicates whether the input voltage is greater than half the reference (2.5V with a 5V reference). Each succeeding bit represents half the range of the previous bit. Table A-1 illustrates this point. Adding the voltages corresponding to each set bit in the word 0010 1100, we get: **.625 + .1563 + .0781 = .859 Volts**

Table A-1 Digital representation of an 8-bit measurement.

Bit	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Volts	2.5	1.25	0.625	0.3125	0.1563	0.0781	0.0391	0.0195
Output Value	0	0	1	0	1	1	0	0

The word 1111 1111 is the number 255 which corresponds to 5*255/256=4.985 Volts. Hence, we cannot represent the reference voltage exactly, regardless of resolution used.

The converter in this project uses 12-bits, rather than 8 bits. A 12-bit word has 4096 unique values, thus giving the 12-bit converter 16 times the resolution of the 8-bit converter for the same input range.

B Implementing a PID controller

In theory, feedback control systems are often easily implemented. In reality however, the situation is often more problematic. In this chapter we try to present some hints and guidelines for how a feedback controller should be implemented, taking into consideration the physical limitations of the controlled system. We do not want to make the discussion lengthy and will therefore only consider the PID controller. Some of the techniques shown here also apply to the state-space controller.

A PID controller is simplest described by the block diagram found in Figure B.1. The control law U is $U = P + I + D$ where P, I and D are the three parts of the PID controller. The reference is y_r and the three parameters for the controller are K, T_i and T_d.

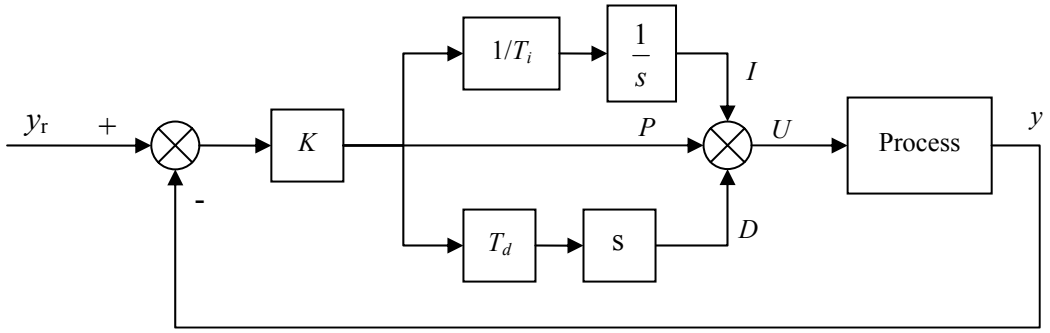


Figure B.1: Classic PID controller.

B.1 Limiting the derivative

Assume the reference y_r follows a step function, i.e. it changes from one value to another during a very short time period. In the controller, the derivative at the time of reference change can become very large. This can disturb the controller to such a degree that it makes the system become unstable. The problem is often solved by using a weight on the derivative part, D. This bounds the maximum derivative of the control signal to a value N.

Instead of using $D = KT_d s(y_r - y)$ as in Figure B.1 above, we use:

$$D = \frac{KT_d s}{1 + \frac{KT_d s}{N}}\left(\gamma \cdot y_r - y\right) \approx N \text{ for high frequencies.}$$

The constant γ is either 1 (servo control) or 0 (process control). Although $\gamma = 1$ often gives a smoother response, the case $\gamma = 0$ is more common (simpler to implement).

B.2 Anti windup

Windup is perhaps the most important factor to deal with when implementing a feedback controller using integration (PID or state space). The phenomenon occurs when there is saturation of the control signal. This happens when the output of the controlled actuator is limited to a certain interval; such as maximum/minimum control angle of a servo, or a maximum/minimum output voltage from a controlled source. It can also occur when the control signal is time dependent, for example servos that can move only a certain distance (degrees) per time unit.

These saturation effects cause the integration part of the controller to misjudge the actual error. This can lead to a state where the controller output starts oscillating without control between the minimum and maximum states. This is commonly known as *windup*.

Since physical devices (actuators) have a limited interval where they can operate, it is obvious that almost any controller should include an anti-windup system. Here follows a proposal for a commonly used anti-windup solution.

Taking the difference between the desired control signal V and the bounded control signal U the integration step can be modified according to the block diagram shown in Figure B.2. T is a new parameter related to the desired amount of anti-windup (T≈10-20).

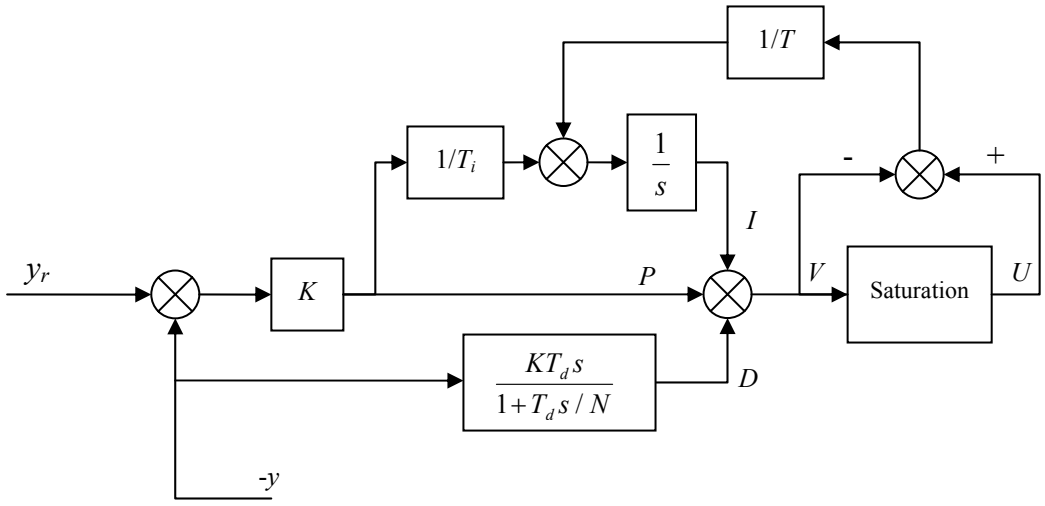


Figure B.2: PID controller with anti-windup and derivative limiter.

B.3 Implementation

Using a shift-operator q^{-1} such that $q^{-1}y(k)=y(k\text{-}1)$, where k is a discrete time step, we can define s of the Laplace transform in an Euler-backward sense using an arbitrary time constant h (equivalent to the reciprocal of the sampling frequency).

$$s \to \frac{1-q^{-1}}{h}$$

We can now calculate the discrete implementation of the controller shown in Figure B.2. For the derivative part D we obtain:

$$D = \frac{KT_d s}{1+T_d s/N}(-y) \Leftrightarrow D + D\frac{T_d}{N}s = -KT_d sy$$

$$D(k) + \frac{T_d}{N}\left(\frac{D(k)-D(k-1)}{h}\right) = -KT_d\left(\frac{y(k)-y(k-1)}{h}\right) = \ldots \Rightarrow$$

$$D(k) = \frac{T_d\left(D(k-1) - KN\left(y(k)-y(k-1)\right)\right)}{Nh+T_d}$$

This can be directly implemented in code:

```
D=Td*(D-K*N*(y-old_y))/(N*h+Td);
```

Notice that the old input signal y(k-1) is stored as a separate variable *old_y*.

Next, we look at the integration-factor *I*, which we obtain directly from Figure B.2.

$$I = \left(\frac{K}{T_i}(y_r - y) + \frac{U - V}{T} \right) \frac{1}{s}$$

Using Euler-backward we will obtain:

$$I(k) = I(k-1) + \left(\frac{k \cdot h}{T_i} \right)(y_r - y) + \left(\frac{h}{T} \right)(U - V)$$

The above is used in various controllers to reduce anti-windup. This can be implemented in code as follows:

```
I=I+(K*h/Ti)*(yref-y);  // Desired Integration Factor
V=P+I+D;                // Desired Output Signal
U=sat(V,Vmin,Vmax);     // Bounded Output Signal
I=I+(h/T)*(U-V);        // Corrected Integration Factor
```

Sat(*V*,*Vmin*,*Vmax*) is a function calculating the saturated (i.e. bounded) signal *U* from a control signal *V* in the interval defined by *Vmin* and *Vmax*.

```
double sat(double V,double Vmin, double Vmax)
{
      return V<Vmin? Vmin : (V>Vmax? Vmax : V);
}
```

Below is a complete code for a PID control loop. It uses two functions AD_IN() and DA_OUT() to receive measurements and signal the actuator. We assume all variables to be of type float or double.

```
for(;;)
{
   y=AD_IN();
   P=K*(yref-y);
   D=Td*(D-K*N*(y-old_y))/(N*h+Td);
   I=I+(K*h/Ti)*(yref-y);
   V=P+I+D;
   U=sat(V,Vmin,Vmax);
   DA_OUT(U);
   I=I+(h/T)*(U-V);
   oldy=y;
   //Wait for next control cycle here
}
```

C Switch control circuitry

When controlling electronic devices using a PC-compatible computer, additional electronic components are usually required. Here we present a circuit diagram for a simple digital switch which can be controlled from a TTL logic connector. On a PC, the pins of the parallel port are TTL compatible and thus can be used to control the switch presented.

In this project, there are three types of components (loads) which we want to control. Their types and electrical characteristics are found below.

Table C-1 Electrical characteristics for the loads.

Load	Required Potential	Required Current	Effect
Solenoid valve (high pressure)	12V	1.5A	18W
Solenoid valve (low pressure)	12V	450mA	5.4W
Glow plug	1.25V	3A	3.75W

Using this information it has been possible to devise a switching circuit for the first two of the three components. The schematics in Figure C.1 is designed to switch a 12V signal on and off for opening and closing the two solenoid valves in Table C-1.

Components required:

- MOSFET (transistor), type P16NE06L from ST electronics
- Diode, type N4007
- 100 Ohm resistor

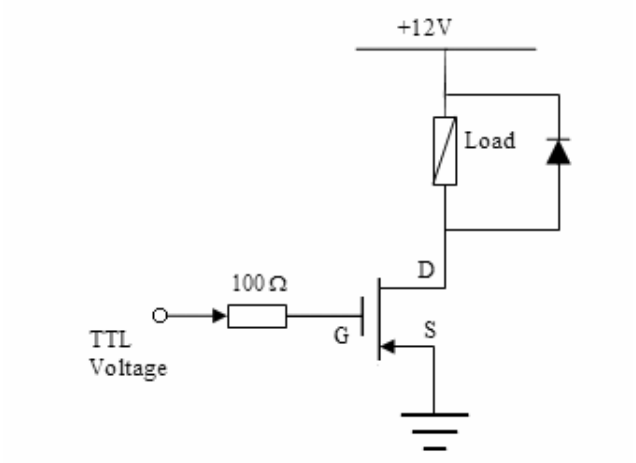


Figure C.1: Circuit diagram.

The diode is used to prevent damage to the FET in the case of load (solenoid valve) disconnection. A similar digital switch can be built for the glow plug once it is possible to transform the source signal to a low-voltage, high-current output signal according to the specifications in Table C-1.

A TTL signal is either 0V which resembles a digital zero, or +5V which is equivalent to a digital one. Changing the states of the bits corresponding to the pins of the digital I/O port will thus open the valve when the state is one, and close it when it is zero.

Important notice about the parallel port

The PC/104 computer leaves all the pins of the parallel port in a high state (digital one) during the first seconds of power up. If solenoid valves (and/or glow plug) are connected it will cause all solenoid valves to open (and glow plug to heat), which may cause undesired effects. The data acquisition card connected to the PC/104 has two digital ports similar to the parallel port. Using one of these ports (which remains at a low state during power up) will solve the problem. Programmatically, these ports are as easy to use as the parallel port. See the file *ioports.h* for information on how to control them.

D Using QNX 6.2

QNX is designed to behave like any UNIX environment. The differences are found in the internal structure and implementation of the operating system itself. This appendix serves as a quick introduction on how to use and develop for the QNX OS.

D.1 Mini how-to

For learning more about how to use QNX, any good book on UNIX or LINUX will do, since most of the material also apply to QNX. In order to immediately get started with QNX, some hints will be presented in this section. The first two subsections are common for all UNIX systems while the others are specific for QNX.

D.1.1 Often used console commands

ls -al – Lists all files in the directory including sizes and permissions.
ps – Shows all running user processes and their corresponding process id (pid).
kill – Terminates a running process by giving the pid (see above) as an argument.
& – Can be put at the end of the line when issuing a command or starting a program. The program will run in the background, as a separate process, (i.e. not "locking" the terminal window).
"./" – In order to execute a program located in the same directory, put "./" before the name of the executable (e.g. "./myprogram").
Ctrl-c – Aborts a program running in the terminal window.
use – Displays help information about a given command or program.

D.1.2 Changing permissions for a file to become executable

Type:

```
chmod 775 [filename]
```

D.1.3 Automatically start programs at system boot

From Photon (the QNX windows environment), open a terminal window and type:

```
ped /etc/rc.d/rc.local
```

The Photon editor will start.
Type the desired commands/programs in the text file.
Save the file.
From a terminal window, type:

```
cd /etc/rc.d
chmod 775 rc.local
```

The commands in rc.local will now be executed at boot.

D.1.4 Installing QNX 6.2 on the PC/104 using the desktop computer

These steps should only be performed if the installation on the PC/104 has stopped working.

Remark: If there are two hard drives installed in the desktop computer, exchange all references to "hd1" in the commands below, to "hd2".

With a switched off desktop PC, insert the Compactflash card from the PC/104 into the desktop PC card holder. Make sure the card holder is connected to the computer's IDE interface. Boot QNX on the desktop PC. On questions about which libraries to bind, make sure it is the local disc (hd0). Usually the combination to press is "F2" then "F1".

Once logged in, open a terminal window and type:

```
dinit /dev/hd1
fdisk /dev/hd1
```

In fdisk, delete all old partitions on the CompactFlash card and add a new one with type 77 (QNX4). Make sure the partition is set to bootable (a '*' should be present under *boot*). Save the settings and close fdisk. A new device /dev/hd1t77 should now exist. Now type:

```
dinit -h /dev/hd1t77
```

QNX automatically mounts all drives under the directory "/fs" (filesystem). There should be a directory called /fs/hd1-qnx4, this is where all files will go. Using the *Filemanager* (found in Photon), locate the following directory:

/fs/hd0-dos-2/Program Files/qnx/boot/fs/

In this directory, there should be a number of files:

```
qnxbase.ifs
qnxbase.qfs
qnxbasedma.ifs
root.qfs
```

Remark: On some QNX systems, these files are located in /boot/fs/.

Copy qnxbase.ifs and qnxbase.qfs to:
/fs/hd1-qnx4/
Using the Filemanager, create the folder:
/fs/hd1-qnx4/boot
and then create the folder:
/fs/hd1-qnx4/boot/fs
move qnxbase.qfs to this directory by typing in the terminal window:

```
cd /fs/hd1-qnx4
mv qnxbase.qfs ./boot/fs/
```

Then type:

```
mv qnxbase.ifs .boot
```

A .diskroot file is missing. The easiest way to retrieve one is to copy the .diskroot file from the desktop PC by typing:

```
cp /.diskroot
```

(from the same directory as before)

Shut down the desktop computer, remove the flash-card from the holder, insert it into the PC/104 and power it up. The PC/104 should now boot by itself.

D.1.5 Disabling Photon boot

QNX automatically boots into the Photon windowing system. To prevent this do the following:
From a terminal window, type:

```
touch /etc/system/config/nophoton
```

To automatically boot back into Photon again, simply remove the file *nophoton*.

D.1.6 Setting up the network card

The easiest way to configure the network settings is to use Photon. Click on the network button from the launch bar, a new window will appear. QNX offers two types of settings:

Manual, which lets the user enter an IP address and DHCP which dynamically sets the address. Note that the QNX computer needs a connection to a DHCP enabled network for the latter to work.

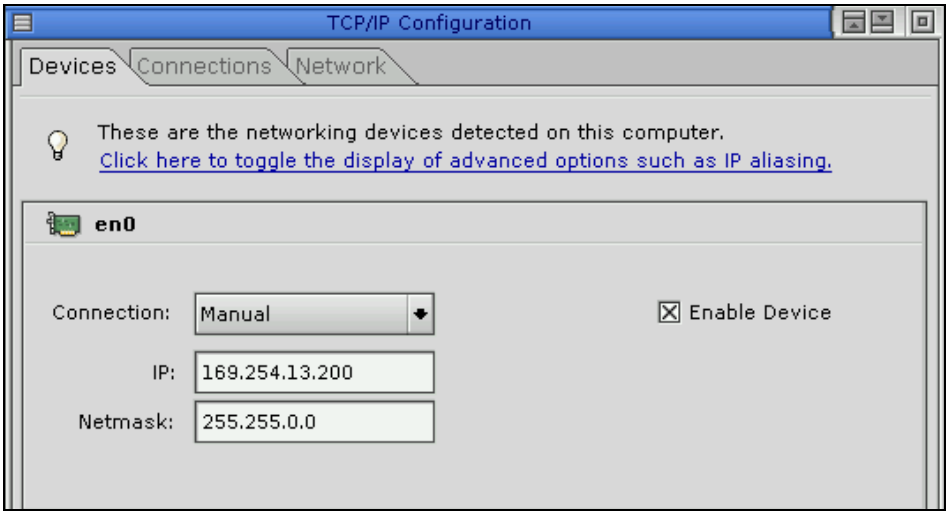


Figure D.1: TCP/IP configuration.

D.2 Software development in QNX

QNX does not offer an integrated development environment, at least not in the non-commercial version. Using the Photon windowing environment makes the development process a lot easier. A typical working session consists of a number of open windows and programs. Using a text editor it is possible to write code. A terminal window can further be used to compile the code. An extra terminal window running sftp (secure ftp) or ftp is used in order to copy the compiled program to the PC/104. Finally, a third terminal window running ssh (or telnet) with an open connection to the PC/104 is used to execute and test the programs. Testing the programs on the PC/104 is necessary if it uses the data acquisition card, servo controller or other special hardware.

The Photon windowing environment offers an excellent documentation and help facility. To launch it, click on “Help”, either from the launch menu or the launch bar. Here, everything there is to know about programming in QNX can be found, including definitions, examples and articles.

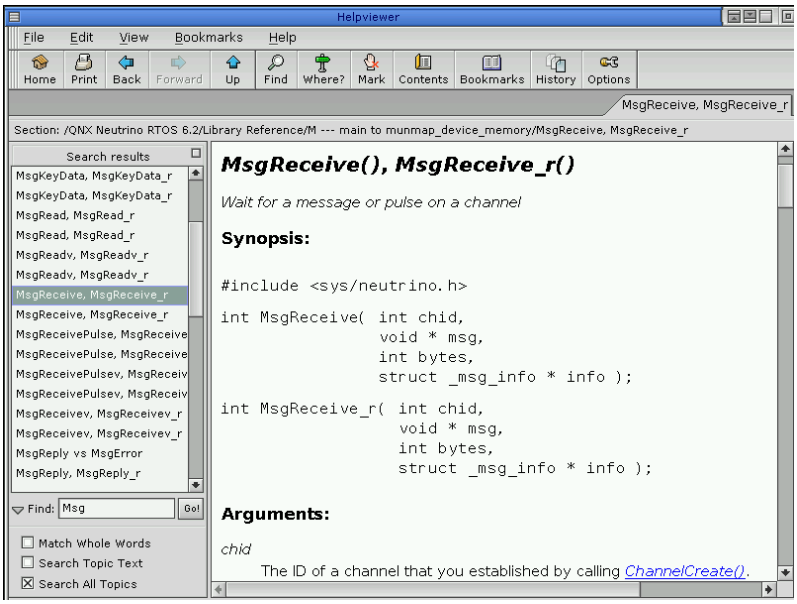


Figure D.2: The QNX help viewer.

D.2.1 Programming in QNX

The only text editor that comes shipped with QNX 6.2 non-commercial version is the Photon editor. The Photon editor is influenced by the Notepad editor found in Microsoft Windows. It offers about the same level of programming capabilities, which is close to none. A better development environment is preferable but there are no alternative editors at the time of this writing for the QNX Photon windowing environment.

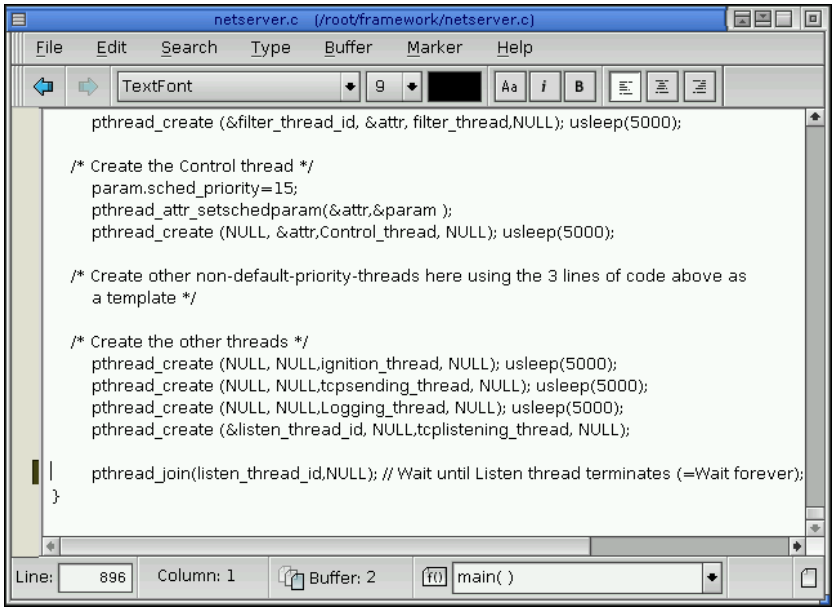

netserver.c (/root/framework/netserver.c)
File Edit Search Type Buffer Marker Help
TextFont
9
pthread_create (&filter_thread_id, &attr, filter_thread,NULL); usleep(5000);
/* Create the Control thread */
param.sched_priority=15;
pthread_attr_setschedparam(&attr,&param);
pthread_create (NULL, &attr,Control_thread, NULL); usleep(5000);
/* Create other non-default-priority-threads here using the 3 lines of code above as a template */
/* Create the other threads */
pthread_create (NULL, NULL,ignition_thread, NULL); usleep(5000);
pthread_create (NULL, NULL,tcpsending_thread, NULL); usleep(5000);
pthread_create (NULL, NULL,Logging_thread, NULL); usleep(5000);
pthread_create (&listen_thread_id, NULL,tcplistening_thread, NULL);
pthread_join(listen_thread_id,NULL); // Wait until Listen thread terminates (=Wait forever);
}
Line: 896
Column: 1
Buffer: 2
main( )

Figure D.3: Photon Editor.

D.2.2 Compiling C programs

The C compiler in QNX 6.2 is called *qcc*. It is a wrapper executable for the Gnu C compiler *gcc*. Gcc users will have no problems using qcc since the usage of the two compilers is equivalent. Below is an example of a normal compilation command:

```
qcc netserver.c -o netserver -l /lib/libsocket.so
```

The program above utilizes an external library. If possible, use .so libraries since they are dynamically linked libraries (generate small executables). To learn more about the C compiler, find a good reference on the gcc compiler.

D.2.3 Using a remote terminal to access a QNX computer

QNX computers only require the user to be physically present when working with a graphical environment. For text or terminal mode it is possible to work from a remote connection. The most common remote terminal program is called telnet and is installed on almost any platform. Another program is called *ssh*. Unlike telnet, ssh uses an encrypted (secure) session. *Openssh*, a free ssh program has been installed on both the desktop QNX computer as well as the PC/104 computer. To start a session from the desktop computer to the PC/104, type:

```
ssh 169.254.13.200
```

or

```
./connect
```

The latter is just an executable macro found in the root directory containing the prior line.

D.2.4 Copying files between two computers using ftp

Almost any computer has a program called ftp, which can be used to transfer files between two computers. The Openssh package contains *sftp*, which is a secure ftp program.

To start an ftp session from the desktop QNX computer to the PC/104, type:

```
sftp 169.254.13.200
```

or

```
./connectftp
```

The latter is again an executable macro saving the user from typing the IP address of the PC/104.

Ftp is a text-based program which lets the user send and retrieve files from the current directory. If the user wants to transfer files from a subdirectory, say from /root/framework/ he or she can type the following:

```
cd /root/framework
../connectftp
```

Files can be transferred from the current directory to the remote computer by typing:

```
send [filename]
```

To copy a file from the remote computer to the local directory type:

```
get [filename]
```

To close the program, type ‘quit’ or ‘exit’.

E Software framework for autonomous vehicles

In this appendix, we discuss the C programs developed for this project. The code itself (which can be found in Appendix G) forms a framework for future development.

In order to program in QNX proper understanding of the C language, POSIX, real-time programming and knowledge of the QNX API is required. A *must read* before trying out any programming in QNX, is the book by Krten [Krt99] which explains both the basics of real-time programming as well as QNX programming. Learning how to perform low-level API calls directly in C is essential in time critical systems such as this. High-level languages such as C++ should be avoided if possible since it is harder to analyze the timing overhead of the compiled code.

E.1 File structure

The framework consists of a number of C source and header files. Basically there are two programs that execute on the PC/104 during operation. Firstly, there is the data acquisition driver (resource manager), which controls the DAQ card and allows other programs to use it. Secondly, it is the control program itself. This is a single C program containing all subsystems discussed in Chapter 3.

Table E-1: Header files.

dmadef.h	Definitions used by the data acquisition driver to access the DMA hardware.
IDList.h	Defines macros used both for internal and remote communication.
ioports.h	Definitions and (pseudo-)functions for access to the parallel port as well as the two digital I/O ports located on the DAQ card.
pcm3718.h	Definitions for the data acquisition driver.
pulsecodes.h	Definitions of the pulse codes used by the framework.
servocontrol.h	Code used to access the servo controller.

Table E-2: Source files.

netclient.c	A text client which can be compiled both in QNX as well as in LINUX. Can be used as a base for a more sophisticated control and monitoring programs of the PC/104.
netserver.c	This is the main program. It compiles in QNX but will only execute on the PC/104 computer.
pcm3718.c	This is the software driver (resource manager) for the data acquisition card. Only runs on the PC/104.

The program *netserver.c* is the main program of this project. The program is not complete and it needs several hours of work before it can perform any autonomous control. It consists of almost 900 lines of code, from which new applications can be built. The software driver (or resource manager) is complete and ready to be used in any application requiring a DAQ module of the same series as the one used in this project.

E.2 Threads and the C language

Each running program in QNX is considered to be a process with at least one thread. In C code, a thread is a real-time element that is syntactically equivalent to a normal C function.

Instead of being called as a normal function it can be initiated to execute concurrently with other threads of the same process. If desired, several instances of the same thread can be executed without risk of error.

An interesting thing with threads is that they share memory with other threads of the same process. In practice this means that all global variables in the C source file are shared between threads the same way they are shared in a normal C program. Caution must be taken, though, since threads are running concurrently data protection must be used in order not to corrupt the shared data. Such protection systems are integrated as part of the QNX operating system.

E.2.1 Thread overview

The *netserver.c* file is divided into a number of threads. A good design strategy is to have many threads, each performing a specific task. Having a small number of threads performing multiple tasks is often more error prone.

Below is the thread layout of the program *netserver*.

1. **A/D Filter thread** – This is the only thread in the program that has access to the data acquisition card. It sits like a gateway between the card and the rest of the real-time system and converts raw data (ones and zeros) produced by the data acquisition card into real numbers such as voltages and currents. It also filters the incoming data from measurement noise.

2. **Attitude control thread** – This thread is basically a feedback control loop. It acts on input from the inertial measurement unit and guides the rocket by moving servo motors connected to external fins attached to the rocket body. The thread is not yet implemented.

3. **Rocket ignition and thrust control thread** – The rocket engine is controlled by the opening and closing of onboard solenoid valves, which in turn are controlled from this thread using digital switching circuitry. In the current implementation, the code for this thread is not implemented. This will have to wait until the electrical system for the engine control has been completely designed.

4. **File logging thread** – Saves specified data to local file(s) onboard the PC/104.

5. **TCP/IP Listen thread** – The feedback communication system uses TCP/IP to connect to a remote computer over the Internet or a local area network. The Listen thread receives and acts upon incoming data such as commands sent by an operator.

6. **TCP/IP Send thread –** Like the Listen thread, the Send thread uses TCP/IP as communication method. It sends feedback data to the remote computer for operation maintenance and analysis.

7. **Heartbeat thread –** This thread also uses TCP/IP. It sends and receives connection status data in order to prevent data loss.

8. **Startup thread –** For a real-time program to become multithreaded it must begin as a single thread. This thread is the C-programs *main()* function. It basically starts all other threads and then waits indefinitely, allowing the other threads to utilize the CPU.

E.3 Data interchange

As stated earlier, all threads in a process share the same memory. Figure E.1 shows how this memory model is used to transfer measurement data between threads.

The “Raw data”, a memory area shared between the data acquisition software driver and the Filter thread is converted to meaningful values by the filter. These values are placed in a separate memory area (“Measurement data”), located inside the process (the *netserver* program) for global access by other threads in the process/program.

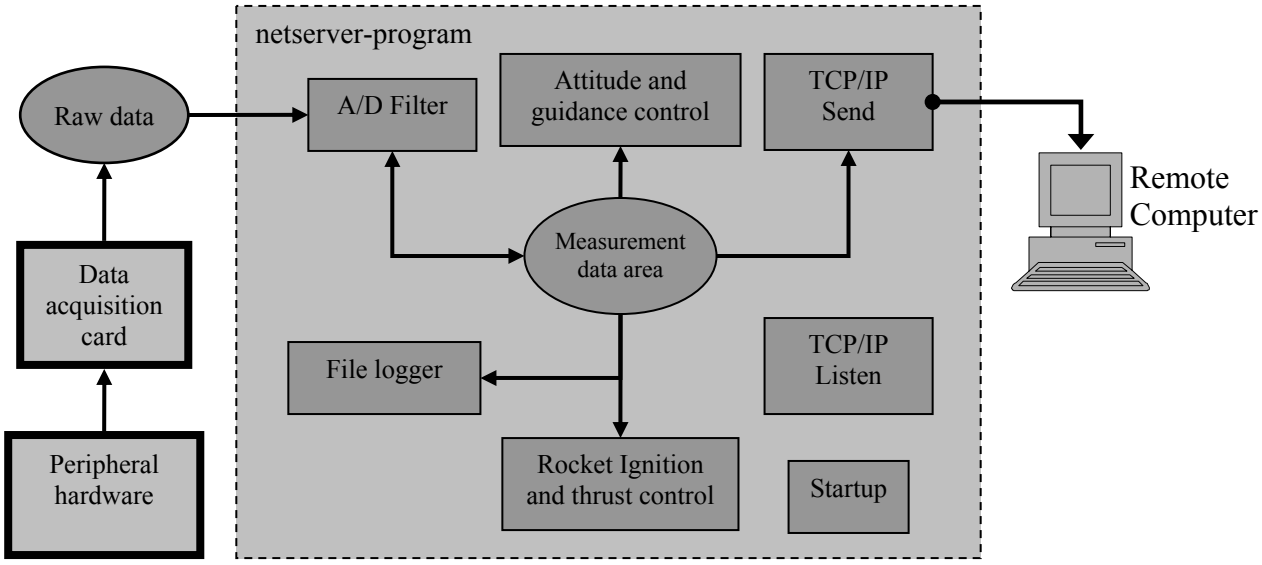


Figure E.1: Overview of measurement dataflow in and out of the real-time control system.

E.4 Platform communication

The framework supports communication using TCP/IP network. In such networks, one side acts as a host or server and the other acts as a client. In this project, the PC/104 is acting as the network server. When starting the *netserver* program, it waits for a connection from a remote client over the network. When connection is established, the client can start sending commands to the server such as telling it to start and stop A/D conversion or to move servos. The server can also send data back to the client for monitoring.

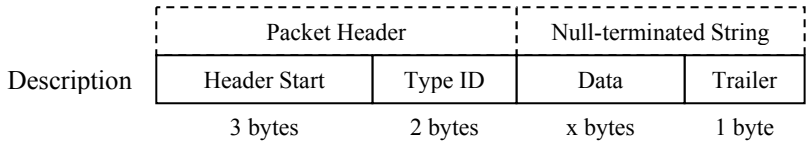
E.4.1 Communication protocol

In the lower layers of TCP/IP data is sent using small packets. This model is utilized in the application layer as well. The TCP/IP send and receive threads have sophisticated architecture to pack and unpack data inside these application layer packets.

E.4.1.1 Application layer packet description

Each packet starts with a five byte packet header. The header always begins with a three byte character sequence defined by the constant character array *HeaderStart* (defined in *netserver.c*). The start sequence is followed by a two byte *Type ID* number, which identifies what type of data is contained in the packet.

After the packet header follows the data, it consists of a null-terminated character array, or C-string. The string can be of any size and the null character acts as a packet trailer.



Note that none of the characters in the data is allowed to be the null character. This limitation requires all internal types such as *int*, *float* or *double* to be converted into plain (ASCII) text before it is sent over the network. If the data is not converted there is a risk of a null character entering the data string, ending the packet prematurely. Standard functions such as *sprintf* can be used for this purpose.

E.4.2 Type IDs

The Type ID is a simple unsigned 16-bit value (0-65535) identifying what kind of data the packet contains. The number can for example represent a variable value or a command. C-macros are used to represent these numbers. These are defined in the file *IDList.h*.

An example of definitions in this file could look like this:

```
// A/D channels
#define       ID_USER       10
#define       ID_CHAN0      ID_USER+0
#define       ID_CHAN1      ID_USER+1
#define       ID_CHAN2      ID_USER+2
#define       ID_CHAN3      ID_USER+3
#define       ID_CHAN4      ID_USER+4
#define       ID_CHAN5      ID_USER+5
#define       ID_CHAN6      ID_USER+6
#define       ID_CHAN7      ID_USER+7

// Client Commands:
#define ID_START_AD         1024
#define ID_STOP_AD          1025
```

The numbers between 0 and the macro ID_USER (i.e. numbers 0 through 9) are reserved for the internal use of the framework. The file defines 8 analogue channels that can be used by the PC/104 in order to send measurement data back to the remote client for data collection and/or analysis. The remote client can also send two commands to the PC/104 server. As a convention, command identifiers start at 1024.

E.4.3 Type ID interpreter

When data packets arrive over the network to the server program, the 16 bits containing the Type ID (type identifier) needs to be converted into a format which it can understand and act upon. This is automatically performed by the TCP/IP Listen thread. By comparing the received number with the macros defined in IDList.h using a *switch* statement, the Listen thread can both decipher and act upon the command or relay it to other threads. A minimal command interpreter can be found below:

```
switch(ntohs(PacketID.asUintN))
{
case ID_START_AD: // this macro is defined in IDList.h
       MsgSendPulse (chid_filt,pulseprio,COMMAND_PULSE_CODE,ID_START_AD);
       MsgSendPulse (chid_send,pulseprio,COMMAND_PULSE_CODE,ID_START_AD);
break;

case ID_STOP_AD: // so is this macro
       MsgSendPulse (chid_filt,pulseprio,COMMAND_PULSE_CODE,ID_STOP_AD);
       MsgSendPulse (chid_send,pulseprio,COMMAND_PULSE_CODE,ID_STOP_AD);
break;
}
```

This interpreter treats two incoming commands, ID_START_AD and ID_STOP_AD, as defined in *IDList.h*.

The structure *PacketID* contains the value of the ID in network byte order, the function *ntohs()* is used in order to convert from network byte order to host byte order. In

order to fully understand how the Listen thread interprets these commands, a discussion of thread synchro-nization is required.

E.4.4 Thread synchronization

The QNX operating system offers both the POSIX synchronization solutions as well as its own. The framework uses solutions from both of these.

Table E-3: **Synchronization solutions used in this project.**

Solution	Variety of	Defined in
Pulse	Message passing	QNX
Read/Write lock	Shared variable	POSIX

E.4.4.1 Pulses

Processes normally synchronize using message passing. Normally they are time consuming operations, transferring big chunks of data. A QNX pulse is a lightweight message using a fixed size. They can be used to notify a thread or process that a certain event has occurred or to transfer very small chunks of data (maximum of four bytes). Pulses are used for these purposes throughout the framework.

E.4.4.2 Read/Write locks

When multiple threads are accessing the same data area, such as a structure or an array, there is a risk that one thread will write to the area at the same time as another thread is reading from it. This could cause data corruption to occur. Consider the layout of Figure E.1, where the filter is constantly updating the measurement data area whilst a number of other threads are reading from it. To prevent data corruption, we must use some form of mutual exclusion. A read/write lock offers this capability by allowing only one thread to write to the area while no other thread is reading from it. If no thread is writing to it, however, multiple threads can access the memory area for reading, increasing throughput.

E.4.5 Message passing

An overview of the message passing and control of the system can be seen in Figure E.2. Most threads in the real-time platform are nothing more than simple message dispatchers. This solution is very convenient, since the thread can do synchronized tasks, such as reacting to a timer, as well as listening for incoming (asynchronous) messages from other threads. This is so because both QNX timers and the internal communication system use pulses.

Before pulses can be received, a thread needs to create its own communication channel. Each channel has its own channel identifier number or simply, *chid*. Other threads use these chids when they want to send a pulse to a specific thread. Each thread that requests messages from other threads create their channels in global namespace (of the C program). Other threads can then connect to these channels using the corresponding chid.

As an example, a table of chid names used by the listen thread is found in Table E-4.

Table E-4: Listen thread channel names.

Channel identifier name in global namespace	Connection name in Listen thread namespace	Connects to
channelid_filt	chid_filt	Filter thread
channelid_send	chid_send	Send thread

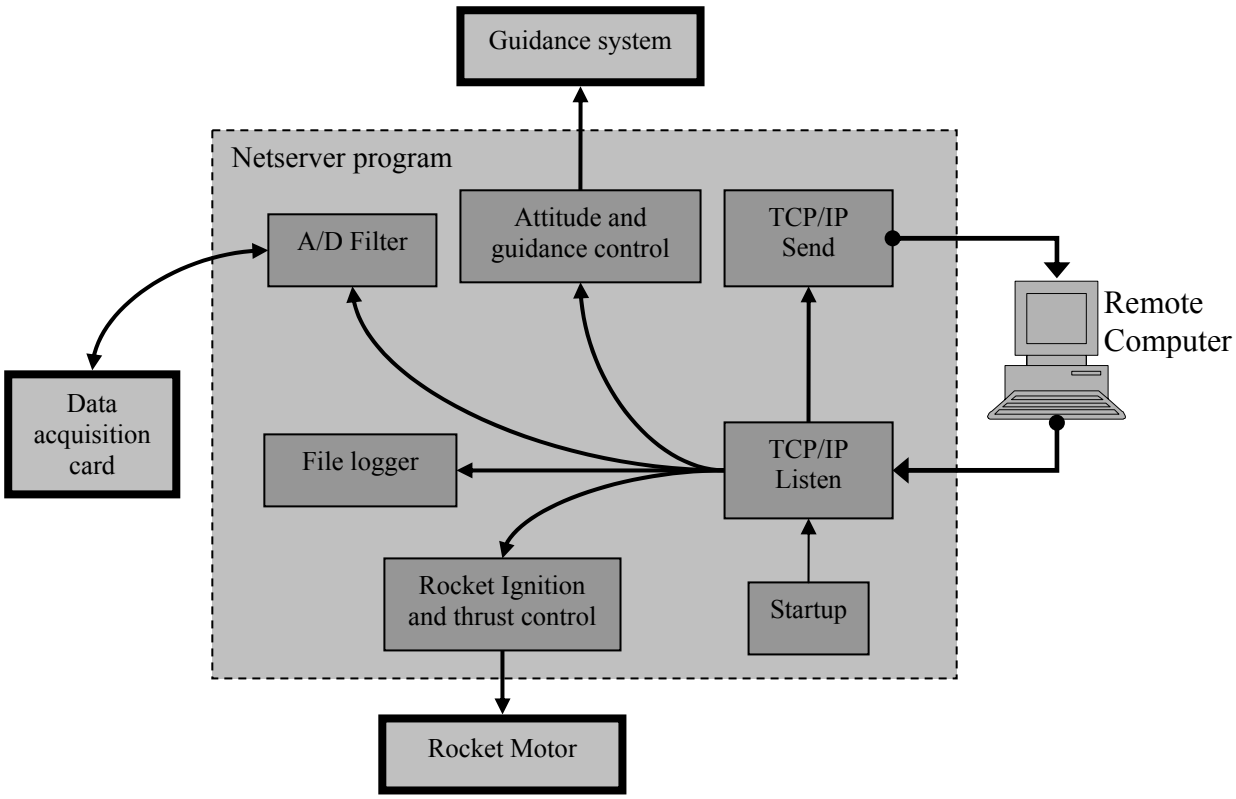


Figure E.2: Framework communication flow.

E.4.6 Remote command relaying

We are now ready to go back to the Listen thread command interpreter. The case ID_START_AD looked like this:

```
case ID_START_AD:
       MsgSendPulse(chid_filt,pulseprio,COMMAND_PULSE_CODE,ID_START_AD);
       MsgSendPulse(chid_send,pulseprio,COMMAND_PULSE_CODE,ID_START_AD);
break;
```

The exact same pulse is being sent to two different channels, one to *chid_filt* (Filter thread), and one to *chid_send*, (Send thread).

Besides the connection identifier, each pulse has a priority, a pulse code and a value to be entered in that order when *MsgSendPulse()* is issued. The *COMMAND* pulse code is used in order to tell the particular thread that a remote command has been received (or locally issued). The command type is passed in the value field of the MsgSendPulse() function. Note that the ID List macro is reused here. Thus the ID List defined in *IDList.h* is used both for network communication as well as internal communication. Other pulsecodes used are defined in the file *pulsecodes.h.*

E.4.7 Sending measurement data over the TCP/IP connection

When an ID_START_AD command pulse is received by the Send thread, it starts sending measurement data to the remote client. In the current implementation it sends measurements every second, controlled by a timer.

The thread uses a wrapper function to add a measurement packet to the current TCP/IP send buffer. The essential code is found below:

```
pthread_rwlock_rdlock(&ad_data.rwl);
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN0,ad_data.channel[0]);
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN1,ad_data.channel[1]);
pthread_rwlock_unlock(&ad_data.rwl);
```

Since the measurement data area (see Figure E.1) uses a read/write lock, it first needs to gain read access to it. It then adds A/D channels 0 and 1 to the send buffer and finally unlocks the read/write lock so other threads can access the area.

Adding measurement channels is fairly easy. For example, if something interesting is measured on A/D channel 3, then in order to send snapshots of this channel to the remote client, add a line with the new channel number inside the Send thread, like this:

```
pthread_rwlock_rdlock(&ad_data.rwl);
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN0,ad_data.channel[0]);
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN1,ad_data.channel[1]);
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN3,ad_data.channel[3]);
pthread_rwlock_unlock(&ad_data.rwl);
```

E.5 Remote platform control

The platform can be remotely controlled over the TCP/IP network. In order for this to work the following is required:

1. The necessary Type IDs have been defined in *IDList.h*.
2. The Listen thread must have a specific case statement interpreting the Type ID.
3. The remote client has a list of all Type IDs the platform can handle.

Remote communication requires at least two computers. If one computer updates its ID List, it will not automatically get updated on the other. This requires constant updates in both host and client programs to make sure both ID List tables are identical. To avoid this problem the framework supports automatic retrieval of the ID List from the server. This means that maintenance of the ID List is required only on the server (PC/104).

E.5.1 QNX and Linux client

A simple client program has been developed, it compiles in QNX as well as in Linux.
Unfortunately it does not utilize the automatic retrieval system to get the ID List. This requires that the same ID List needs to be included when compiling both the server and the client. This is usually not a problem, since development of both the client and server programs is done on the same computer.

The client is called *netclient* and its corresponding source is *netclient.c*. It is a simple text-mode client. It can send a sequence of commands to the PC/104 server when started. It then prints all incoming messages from the server, including the Type ID number and the data.

```
$./netclient 169.254.13.200
Received 26 bytes of data
ID=0    Value=STARTING A/D Conversion
Received 30 bytes of data
ID=10   Value=09.997559
ID=11   Value=09.997559
Received 30 bytes of data
ID=10   Value=09.997559
ID=11   Value=09.997559
Received 30 bytes of data
ID=10   Value=09.997559
ID=11   Value=09.997559
Received 30 bytes of data
ID=10   Value=09.997559
ID=11   Value=09.997559
Received 30 bytes of data
ID=10   Value=09.997559
ID=11   Value=09.997559
```

Figure E.3: Linux session running TCP/IP client.

E.5.2 Windows client

A Win32 client has also been developed using Microsoft Visual C++. The program has no direct user interface or control. Instead it is designed to be integrated as part of other Windows programs that support Microsoft ActiveX and COM technologies.

One such program is MATLAB for windows (version 6.5). Besides supporting advanced tools for numerical analysis, MATLAB offers a built in graphical user interface (GUI) builder, called *Guide*. The builder can be used to construct custom GUIs by dragging and clicking. It requires basic skills in MATLAB programming and is very easy to learn.

E.5.3 Windows client communication model

The C++ program sits as a proxy between the Windows operating system and MATLAB. The proxy offers a number of methods, or functions enabling MATLAB to communicate with the QNX system. The methods are found below.

Table E-5: Proxy methods.

Name	Parameters	Description
Connect	Internet address of remote host	Connects to the QNX server using TCP/IP network
Disconnect	-	Disconnects the open connection
SendCommand	Type ID and data string	Sends a command (data packet) to the QNX server
KeepAlive	-	Used for communication link safety mechanism (Heartbeat)

In order to inform MATLAB when a message from the server has arrived *event handlers* are used. Event handlers are small MATLAB functions that MATLAB calls automatically when receiving the corresponding event from the proxy. Events sent by the proxy are:

Table E-6: Proxy events.

Name	Parameters	Description	Event handler
DataRecvd	Array of data packets	This event is triggered whenever data has been received from the server	*DataRecvdHandler.m*
ConnectionStatus	Status integer	Informs MATLAB when the connection is open or closed	*ConnectionStatus-Handler.m*
Ping	-	Used by the Proxy to check that MATLAB is responding	*KeepaliveHandler.m*

E.5.4 Communication link safety protocol

Imagine the PC/104 is controlled in real-time using a remote client. Should the client, crash or lockup during operation there is no way for the PC/104 to be notified of the occurrence, unless there is some kind of safety mechanism. Such mechanism has been integrated as part of the software framework. Both the client and host send small messages over the TCP/IP network every second. Should either side fail to send these messages within a certain deadline, the connection is said to be *dead* and appropriate action can be taken. The deadline is currently set to 5 seconds. MATLAB uses the KeepAlive method in order to send these messages to the PC/104 server.

E.5.5 The GUI interface

A simple interface has been created using Guide, see Figure E.4. It is called *rocketclient* and its corresponding source files are *rocketclient.m* and *rocketclient.fig.* It comes with three event handlers: *DataRecvdHandler.m, ConnectionStatusHandler.m and KeepaliveHandler.m;* which handle each of the events in Table E-6. It has *connect* and *disconnect* buttons, which are invoking the *connect* and *disconnect* methods in the proxy.

Commands are sent to the PC/104 server by adding buttons to the GUI which invokes the SendCommand method. Text boxes can also be added to monitor variables of interest, such as the autonomous vehicles position, velocity and acceleration.

E.6 Tutorial – Using the MATLAB client program

To better understand how the communication system works, we will demonstrate how to use and expand the MATLAB program. First, make sure MATLAB 6.5 or later has been installed onto a Windows 2000 or XP computer (A Windows NT machine should work as well). Also make sure the following files are found in a single folder on the computer:

ConnectionStatusHandler.m
DataRecvdHandler.m
findGuiData.m
KeepaliveHandler.m
rocketclient.fig
rocketclient.m
RocketNetworkProxy.dll

E.6.1 Basic operation

If running the program for the first time, register the dll-file by first running *cmd* (Select *run* from the start menu and type *cmd*).
Set the working directory where the files above have been placed, then type:

```
regsvr32 RocketNetworkProxy.dll
```

If successful, start MATLAB.
Make sure MATLAB's current directory is the directory where the files above are located. In MATLAB, type:

```
rocketclient
```

A window similar to the one in Figure E.4 should appear. Open the file *rocketclient.m* and locate the line stating:

```
invoke(handles.COMh,'Connect','169.254.13.200');
```

This is found inside the function *pushbutton1_Callback*. The numbers inside the string is the internet address of the PC/104 computer. During this project, it was set to **169.254.13.200**. Make sure that the address in *rocketclient.m* corresponds to the IP address of the PC/104 computer.

Start the PC/104 computer. If screen and keyboard is connected to it, login using username *root*, and password *root.* When logged in, start the data acquisition driver by typing:

```
./pcm3718 -s &
```

Then start the real-time platform by typing:

```
./netserver
```

Remark: It is also possible to login using a remote connection program such as *telnet* or *ssh*. See Appendix D for details.

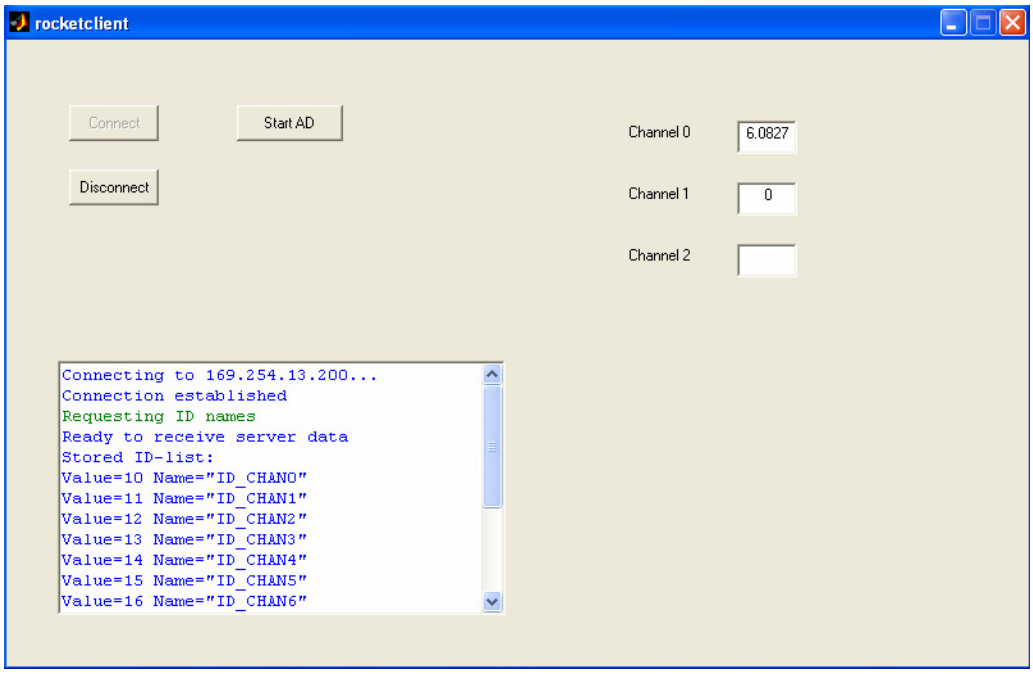


Figure E.4: MATLAB running rocketclient.

Make sure the Windows computer and the PC/104 is connected with each other over a local network using either a network switch or a patch cable. Now press the *connect* button in the MATLAB rocketclient program.

Try pressing the "Start AD" button, the program should behave similar to the one seen in Figure E.4. Numbers should be visible in the first two text boxes on the right side. MATLAB automatically saves incoming data in arrays in the MATLAB workspace with names based on the incoming Type IDs (lowercase with the characters "ID_" stripped away). It is possible to save the MATLAB workspace to file for later use. To close the session, press the *disconnect* button and close the window.

E.6.2 Extending the GUI with a servo controller

We will now demonstrate how to add more functionality to the communication system. Assume we want to control the position of a servo connected to the PC/104 using MATLAB. First, we need to add functionality to the program running on the PC/104. Open the header file *IDList.h* on the desktop QNX computer.

Add a new line in the command section, like this:

```
// Client Commands:
#define ID_START_AD        1024
#define ID_STOP_AD         1025
#define ID_SET_SERVO_0     1026   // <-------- Add this Line
```

Remark: For conformance, make sure all Type IDs start with "ID_".

As discussed previously, the communication protocol supports means of transferring the Type ID names and their corresponding ID numbers. This is done by filling in all IDs currently in

use in a so called *ID Map*. The ID Map is a special structure that is maintained by the netserver program. It sends this map to clients requesting the names and IDs from the server. To get the MATLAB client to recognize the newly added ID, we must add a new entry in the ID Map (located in *IDList.h*). Extend the map with a new IDENTRY, like this.

```
BEGIN_IDMAP
IDENTRY(ID_CHAN0)
IDENTRY(ID_CHAN1)
IDENTRY(ID_CHAN2)
IDENTRY(ID_CHAN3)
IDENTRY(ID_CHAN4)
IDENTRY(ID_CHAN5)
IDENTRY(ID_CHAN6)
IDENTRY(ID_CHAN7)
IDENTRY(ID_START_AD)
IDENTRY(ID_STOP_AD)
IDENTRY(ID_SET_SERVO_0) // <-------------- Add this line
END_IDMAP
```

Save and close the file.

Now open the file *netserver.c* and locate the Listen thread.
Make sure the variable *endptr* has been defined as *unsigned char* * and that an integer named *pos* exists. Also make sure that the function servo_open() has been called, storing the open connection in an *int* named *servo.*

In the Listen thread, locate the Type ID interpreter and add a new case statement:

```
switch(ntohs(PacketID.asUintN))
{
…

case ID_SET_SERVO_0:
pos=strtoul(ptr,&endptr,10);
if (pos>254 || endptr-ptr==0)
{
   printf("Error setting server position : %s\n", ptr);
}
else set_servo(servo,0,pos);
break;
…
}
```

Save the file and compile it from a terminal window by typing:

```
qcc netserver.c –o netserver –l /lib/libsocket.so
```

We now need to transfer the compiled program (*netserver*) to the PC/104 using ftp or sftp (see appendix D).

Remark: If the old netserver program is running on the PC/104, we need to close it first by pressing Ctrl-c. If Ctrl-c is pressed while there is an open TCP/IP connection it is no longer possible to restart the netserver program until a system reboot has been performed. This is a known issue with QNX 6.2.

Now go back to the MATLAB environment. If running, close the rocketclient program and then type:

```
guide rocketclient
```

A new window will open, as in Figure E.5. We are now ready to add new components to the GUI.

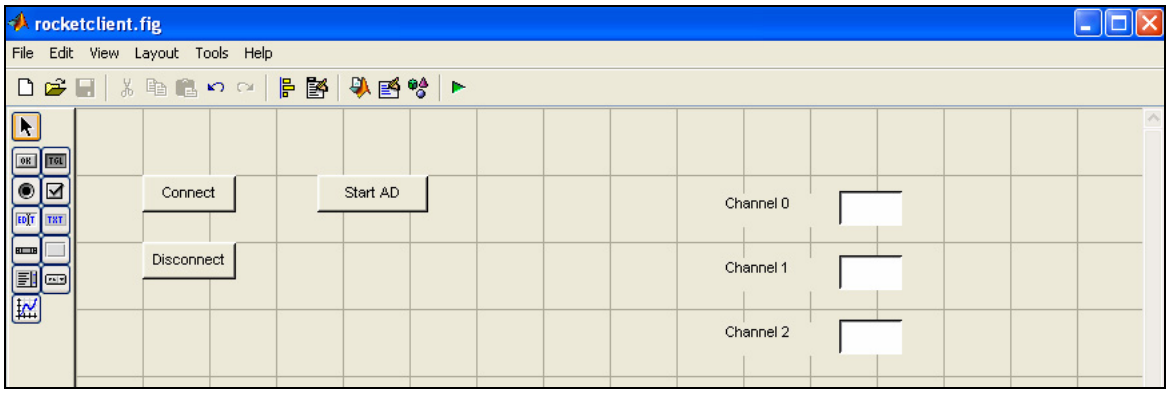


Figure E.5: Guide Window.

From the menu to the left, drag and place a pushbutton and a text box into the GUI area, fairly close to each other, see Figure E.6.

Double-click the new pushbutton; a property window should appear. Find the property name "String" and replace the text with "Set servo". Then find the tag name and change it to "pushbutton4". Finally, close the property window.

Now double-click the edit box. Again, find the property "String" but this time replace it with "0". Next, find the tag and make sure it is called "edit4". Close the property window, then click save and close the Guide window.

Remark: Tag names can in reality be anything, as long as the user remembers them when editing the code in the corresponding m-file.

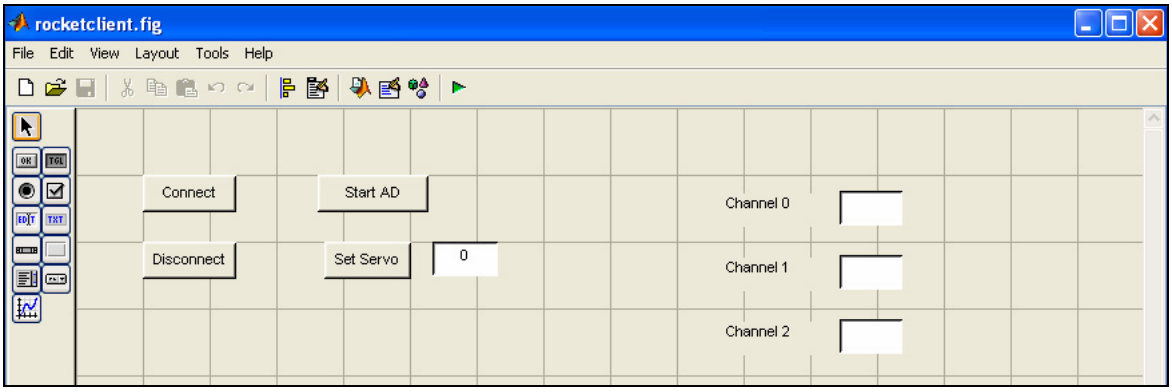


Figure E.6: GUI with servo control.

Open the m-file, *rocketclient.m*. *Guide* has automatically written code for us in this file. Locate the callback-code for the new pushbutton, "pushbutton4". It should look like this:

```
function pushbutton4_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

At the end of this function, add the following line:

```
invoke(handles.COMh,'SendCommand','ID_SET_SERVO_0',get(handles.edit4,'String'));
```

Save and close the file.

Now start the program by typing:

```
Rocketclient
```

Make sure the modified version of the netserver program is running on the PC/104 (following the same procedure described earlier). Push the connect button and wait for connection. Now, try to enter a number between 0 and 254 in the edit box and then push the "Set servo" button. If everything has worked out ok, the servo (if there is one connected on PWM channel 0) should move to the corresponding position.

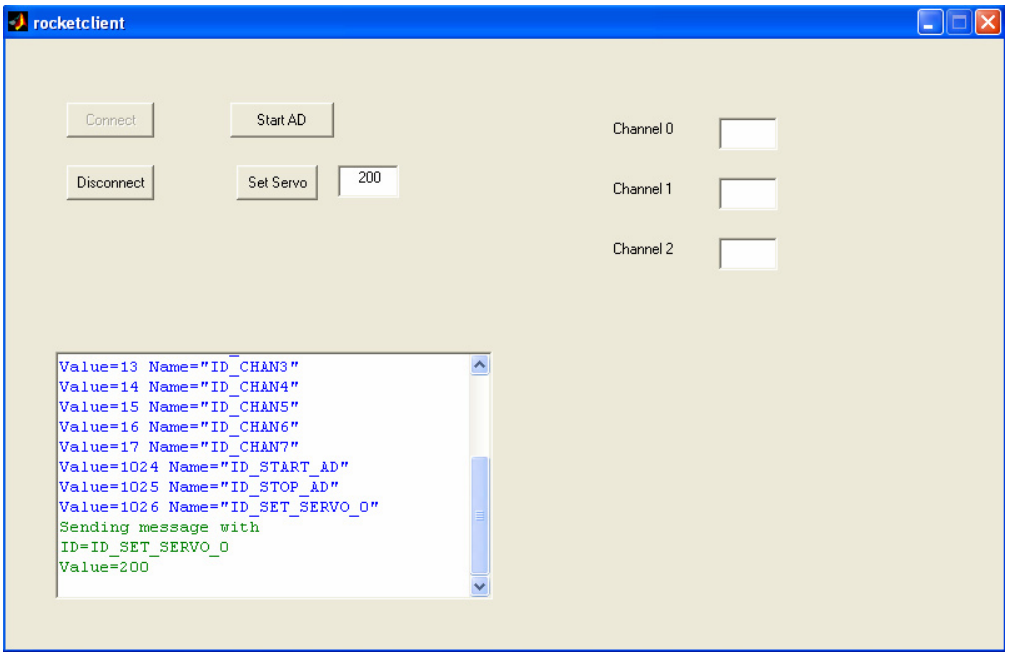


Figure E.7: Rocketclient with servo control.

E.6.3 Expanding the GUI with new measurement displays

The GUI can display data from analogue channels 0, 1 and 2. Assume we want to connect some signal on channel 3, measure it using the DAQ module and then display it inside the MATLAB GUI. As before, we first have to add the functionality on the server side. This time we do not have to add a new entry in the ID List or ID Map, since the first 8 analogue channels already have been defined.

Open the file *netserver.c*.
First make sure the Filter thread is performing A/D conversion on channel 3.
There should be two lines like this:

```
channel_setup_data.start=0;
channel_setup_data.stop=4;
```

Make sure that the stop channel is 3 or more, as above.
Next, find the Send thread, and add the line:

```
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN3,ad_data.channel[3]);
```

inside the message handler for `"TIMER_PULSE_CODE"`, as described in section E.4.7.

Compile and make sure the new program file is running on the PC/104 computer, as described earlier. Now return to MATLAB and start Guide.

```
guide rocketclient
```

Examine the edit boxes on the right-hand-side of the window area by double-clicking on them. Notice that these edit boxes all have property “Enable” set to “Inactive”. This means that the user cannot type in these boxes while the program is running. This is desired, since only MATLAB itself should be able to change the contents of these fields.

The most interesting thing about the boxes is that their tag names are identical with Type ID names defined in *IDList.h*. This of course is no coincidence. In fact, when data arrives through the proxy into MATLAB, the event handler in *DataRecvdHandler.m* automatically scans all the edit boxes for tag names corresponding to the Type ID of the data packet just arrived. If such tag name exists, the handler updates the box with the received value. This means that we do not have to do much work in order to get our data into the GUI.

Start by copying one of the text boxes and place it below the others. Then copy one of the edit boxes and place it next to the copied text box (as in Figure E.8). Edit the string of the text box to read “Channel 3”, or perhaps a name describing the measured signal. Finally, set the tag name of the edit box to “ID_CHAN3”. Save and close the window. Now we are ready to receive and display the new data from the PC/104.

Remark: The MATLAB program stores all data received by the PC/104 whether or not the user chooses to display them. Instead of displaying the data in a box, we can for example plot them in a nice graph, updating it each time new data arrives.

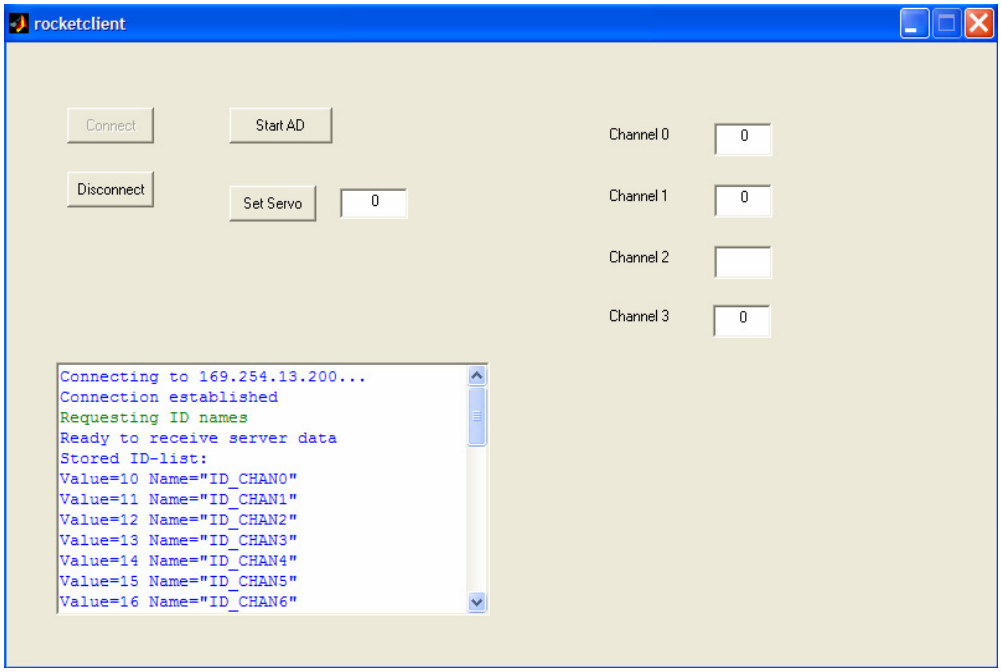


Figure E.8: Rocketclient with new measurement display.

E.6.4 Controlling a solenoid valve

A solenoid valve can also be controlled from the MATLAB computer. Except for the solenoid valve itself, this example requires some extra electrical components set up as a digital switch of an external electrical signal (see Appendix C). If these components exist and have been connected properly, we can start adding software for the switching. Assume we want to control the solenoid valve using one of the pins of the parallel port, say output pin 2.

Start by editing *IDList.h*

Add two new IDs:

```
...
// Client Commands:
#define ID_START_AD                     1024
#define ID_STOP_AD                      1025
#define ID_SET_SERVO_0                  1026
```

```
#define ID_OPEN_VALVE_0                 1027   // <------ Add this line
#define ID_CLOSE_VALVE_0                1028   // <------ And this one
```

Also make two new entries in the ID Map:

```
...
IDENTRY(ID_START_AD)
IDENTRY(ID_STOP_AD)
IDENTRY(ID_SET_SERVO_0)
IDENTRY(ID_OPEN_VALVE_0)                // <------ Add this line
IDENTRY(ID_CLOSE_VALVE_0)               // <------ And this one
END_IDMAP
```

Save and close the file.
Now open *netserver.c* and find the Type ID interpreter inside the Listen thread.
Add two new case statements, adding control of parallel port output pin 2

```
...
case ID_OPEN_VALVE:
       SetPPin(PIN2,1);
break;

case ID_CLOSE_VALVE:
       SetPPin(PIN2,0);
break;
...
```

Save and compile the file. Then start it on the PC/104 computer. Now, go back to the machine running MATLAB and start Guide again by typing

```
guide rocketclient
```

Add two new buttons and edit their text strings to read: “Open valve” and “Close valve”, as in Figure E.9. Also set appropriate tag names. The tag names in this example are “pushbutton8” and “pushbutton9”, respectively.

Save and open *rocketclient.m* and locate the code for the new pushbuttons.
Add the following line for the “pushbutton8” callback function:

```
invoke(handles.COMh,'SendCommand','ID_OPEN_VALVE_0','');
```

and to the “pushbutton9” function, add:

```
invoke(handles.COMh,'SendCommand','ID_CLOSE_VALVE_0','');
```

Remark: When invoking the *SendCommand* method in the proxy we must give both a Type ID and a corresponding data string as parameters. If extra data is not required, as seen above, we simply add an empty string.

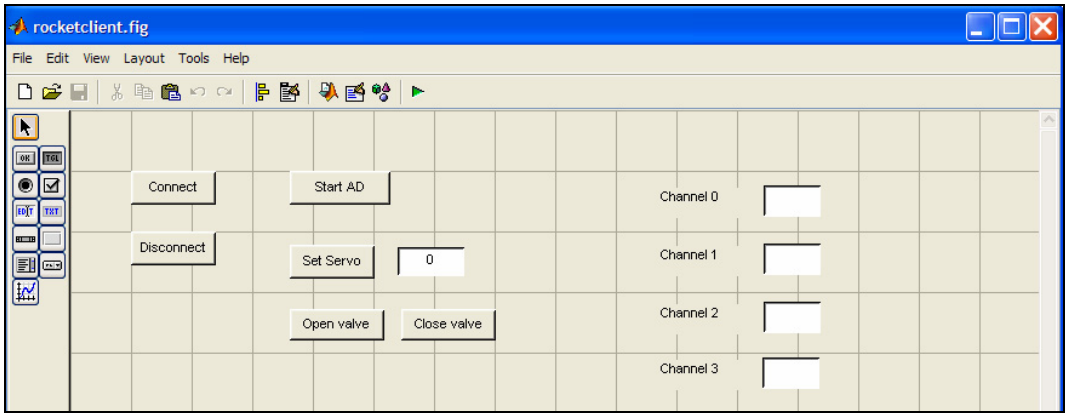


Figure E.9: Guide window with added valve controls.

Save the file and start the program. After pushing the connect button, it should be possible to remotely control the valve.

Remark: The large text window at the bottom left of the GUI does not get displayed in Guide. The window is the Proxy program and since it is not a part of MATLAB it does not materialize until the MATLAB program gets started. When designing the GUI, make sure to leave enough space for this window to appear and do not place any components inside its area, since it will be impossible to access them later.

F Data acquisition software driver

To measure electrical signals, the PC/104 computer has been extended with a data acquisition card. The card used is of model PCM-3718H manufactured by Advantech Co. Ltd. A software driver (or resource manager) has been developed for the QNX 6.2 OS as part of this project. The driver follows the QNX 6.2 guidelines for device driver development and implements the following features:

- 12-bit A/D conversion
- Software A/D conversion (polling)
- A/D conversion using onboard pacer and DMA transfer
- Up to 100,000 samples per second without loss of data
- 16-bit digital input/output
- Data transfer synchronization with client

F.1 Starting the driver

The device driver can be started from the console or during system boot. The DAQ card can be set up in a number of different modes using onboard jumpers. These jumper settings need to be given to the device driver in order for it to function correctly. The device driver assumes default values for all these settings.

The driver program is called *pcm3718* and is found in the */root* directory on the PC/104 computer. it can be run in the three different output modes**.** The mode is chosen when starting the driver, by giving the following arguments.

-n Normal – this means that errors and some general info are printed onscreen to the terminal window it was started from.
-v Verbose – prints a lot of info on the screen telling everything the driver is up to at the moment. This mode is useful when debugging the driver itself but also comes in handy when debugging the program that uses the driver.
-s Silent – does not print any messages to screen. This mode should be used during normal operation.

To change the default driver parameters, use the following arguments followed by the desired value (separated by a white space character).

-b I/O baseadress **default=210**
This is the I/O memory address used by the card. The address is entered using hexadecimal base.

-d DMA channel (1 or 3) **default = 3**
Most PCs support a number of DMA channels (often up to 8) where each hardware device is assigned a separate channel. Channel 3 is often unused and is selected by default.

-c Internal clock frequency (1 or 10 MHz) **default = 10 (MHz)**
The frequency used by the onboard pacer.

-i IRQ (2, 3, 4, 5, 6 or 7) **default = 5**
The software driver uses an interrupt to synchronize the local timer used for data transfers (done approximately 2-4 times / second for normal operation settings). In

reality, this is not a jumper setting, but it is not likely the user will need to change the IRQ value while the driver is running.

-r Transfer rate, default = 1000 (Hz)
This tells how many times per second the driver will notify the client that new data has arrived. This parameter can also be set directly from the client program during card initiation.

Example:
To start the device driver in silent mode, using IRQ 7, 1 Mhz clock frequency and tansfer rate of 2kHz, type:

```
./pcm3718 -s -c 1 -i 7 -r 2000 &
```

F.2 Controlling the DAQ-card

Once the driver is running, it creates a device in ***/dev/PCM3718***.
Client programs open this device by issuing an *open()* command. The client program can then issue control commands to the device driver which sets up the card in a desired fashion.

The commands are sent using the *devctl()* function which is a QNX variant of the *ioctl()* function found in operating systems such as UNIX or Linux.

The devctl function takes 5 arguments:

1. the file descriptor for the open connection (retrieved from *open()*)
2. A control code macro, specifying the operation to perform
3. A pointer to a parameter whose type depends on the control code
4. The size of the parameter
5. Not used, always NULL

There are eight different commands that can be issued in order to control the operation of the DAQ card. The control code macros and their description are found below.

F.2.1 DEVCTL_SETCHANNEL

Parameter type:
channel_setup_data_t (defined in pcm3718.h)

Description:
Sets the voltage input range used for each channel in the DAQ card.
The DAQ card supports either 8 channels in differential mode or 16 channels in single ended mode (jumper setting). The driver supports a unique range for each individual channel. The ranges supported are:

***Bipolar*:** ±10V, ±5V, ±2.5V, ±1.25V, ±0.625V
***Unipolar*:** 0 – 10V, 0 – 5V, 0 – 2.5V, 0 – 1.25V

Example:

```
#include "pcm3718.h"
…
channel_setup_data_t channel_setup_data;
int i,card_fd;
card_fd = open("/dev/PCM3718", O_RDWR);
channel_setup_data.start=0;
channel_setup_data.stop=7;
for (i=0;i<8;i++)
{
  channel_setup_data.range[i]=UNIPOLAR_10V;
```

```
}
devctl(card_fd, DEVCTL_SETCHANNEL, &channel_setup_data,
       sizeof(channel_setupp_data), NULL);
```

This command uses a special data structure containing information such as which channels to include in the A/D conversion loop and the range for each channel used. See pcm3718.h for further information.

F.2.2 DEVCTL_SETPACER

Parameter type:
double_uint (defined in pcm3718.h)

Description:
The DAQ card supports a custom pacer rate (sampling frequency) defined as:

$$PacerRate = \frac{OscillatorFrequency}{C1 * C2}$$

where OscilatorFrequency is either 1 or 10 Mhz as set by jumper 1. C1 and C2 are integer numbers, both between 2 and 65535. These two numbers are passed inside a *double_uint* structure as defined in *pcm3718.h*.

Example:
```
// This example sets the sampling frequency to 100kHz, assuming a 10 MHz clock.
…
double_uint pacerdata;
pacerdata.c1=pacerdata.c2=10;
devctl(card_fd, DEVCTL_SETPACER, &pacerdata, sizeof(pacerdata), NULL);
```

F.2.3 DEVCTL_SET_TRANSFER_RATE

Parameter type:
unsigned int;

Description:
Sets the transfer rate defined by the *int* parameter. This can only be done before issuing the DMA init command. Be aware of that the pacer speed divided by the transfer rate should be an integer for the driver not to loose data.

Example:
```
unsigned int rate;
rate = 2000; // Sets the new speed to 2000 Hz
devctl(card_fd, DEVCTL_SET_TRANSFER_RATE, &rate, sizeof(rate), NULL);
```

F.2.4 DEVCTL_DMA_INIT

Parameter type:
None
Description:
This command can first be issued once the pacer speed has been set. Besides initializing the DMA controller it calculates the sizes and allocates memory for the necessary memory buffers. Once this command has been issued, it is not possible to set the pacer to a new value, or reinitialize the DMA buffers except after first closing the connection to the device and then reopen it.

Example:

```
…
devctl(card_fd, DEVCTL_DMA_INIT, NULL, 0, NULL);
```

F.2.5 DEVCTL_ATTACHFILTER

Parameter type:
attach_message_t (defined in pcm3718.h)

Description:
For the device driver to send messages to a thread in the client program, the client first needs to send enough information to the driver so the driver can find it later. This is done by sending an *attach_message_t* structure to the driver. Unlike all other control commands, the device driver gives a reply to the calling thread telling it where to find the raw data and its corresponding size. The attach_message_t union is big enough to fit both the message and the reply. The device driver synchronously copies data into the shared memory area accessible by the client and notifies the client by sending a pulse. This requires that the calling thread has created a message channel that the pulse can be transmitted in before issuing this command.

Example:

```
attach_message_t fm;
int channel_id, fd;
unsigned char *rawDataBuffer; //The raw-data memory area
…
channel_id = ChannelCreate(0);
fm.data.channelid=channel_id;
fm.data.pidid=getpid();
devctl(card_fd, DEVCTL_ATTACHFILTER, &fm, sizeof(fm), NULL);
fd=shm_open(fm.reply.shm_name,  O_RDWR, 0770))
rawDataBuffer = mmap(NULL, fm.reply.size,PROT_READ | PROT_NOCACHE,
                MAP_SHARED, fd, 0);
```

F.2.6 DEVCTL_AD_START

Parameter type:
None

Description:
Starts the onboard pacer which commences the A/D conversion. This command can only be issued after the following commands have been previously issued:

1. DEVCTL_SETCHANNEL
2. DEVCTL_SETPACER
3. DEVCTL_DMA_INIT

Example:

```
devctl(card_fd, DEVCTL_AD_START, NULL, 0, NULL);
```

F.2.7 DEVCTL_AD_STOP

Parameter type:
None

Description:
Halts the A/D conversion. Issuing a start A/D command later continues the conversion where it left off. When the device is closed it automatically stops the A/D conversion.

Example:

```
devctl(card_fd, DEVCTL_AD_STOP, NULL, 0, NULL);
```

F.2.8 DEVCTL_SET_READMODE

Parameter type:
unsigned char

Description:
The DAQ card supports two 8-bit digital I/O channels. By issuing a *read()* or *write()* command it reads from or writes to a digital port. To perform software A/D the *read()* function is also used. This command selects what operation the read and write function will issue when called from this device. Depending on the value of the *unsigned char* parameter the following relationships can be constructed:

Table F-1 Read-mode overview.

Parameter value	Read From	Write to
RWMODE_DIO1	Digital I/O port 1	Digital I/O port 1
RWMODE_DIO2	Digital I/O port 2	Digital I/O port 2
RWMODE_ANALOG	A/D converter	-

The macros used as parameter values are defined in *pcm3718.h*.

Example:

```
…
unsigned char mode,value;
unsigned int reading;
mode = RWMODE_DIO1;
…
devctl(card_fd, DEVCTL_SET_READMODE, &mode, sizeof(mode), NULL);
value=0xF0;
write(card_fd,&value,1);   // write 0xF0 to Digital port 1
mode = RWMODE_ANALOG;
devctl(card_fd, DEVCTL_SET_READMODE, &mode, sizeof(mode), NULL);
read(cardfd,&reading,2);   //Make a software A/D.
```

Remark: A better and even simpler method to read and write to the digital I/O ports is to use the definitions in ioports.h. This devctl command exists only to provide meaningful operations for the read and write routines, offering an alternative interface for accessing the ports.

G Source code

This section presents the C source code used for this project. An index of this chapter is found below in order to help the reader find the different subsections.

G.1 dmadef.h 90
G.2 IDList.h 91
G.3 ioports.h 93
G.4 pulsecodes.h 94
G.5 servocontrol.h 95
G.6 pcm3718.h 96
G.7 pcm3718.c 99
G.8 netserver.c 116
G.9 netclient.c 129

G.1 dmadef.h

```
/****************** DMA SPECIFIC MACROS**********************/

/* This file is used by the PCM3718 driver to gain access to the INTEL
   DMA controller that sits onboard all modern PC's
*/

unsigned int pageRegs[] = {0x0087,0x0083,0x0081,0x0082,0,0x008B,0x0089,0x008A};

unsigned int countRegs[] = {0x0001,0x0003,0x0005,0x0007,0,0x00C6,0x00CA,0x00CE};

unsigned int baseAddrRegs[] = {0x0000,0x0002,0x0004,0x0006,0,0x00C4,0x00C8,0x00CC};

#define PAGE_ADDR_reg(ch)                       (pageRegs[ch])
#define COUNT_reg(ch)                           (countRegs[ch])
#define BASE_ADDR_reg(ch)                       (baseAddrRegs[ch])

#define CLR_BYTE_PTR_reg(ch)                    ((ch < 4 )? 0x00C : 0x00D8)
#define MODE_reg(ch)                            ((ch < 4 )? 0x00B: 0x00D6)
#define SNGL_MASK_reg(ch)                       ((ch < 4 )? 0x00A : 0x00D4)

#define DEMAND                                  (0<<6)
#define SINGLE                                  (1<<6)
#define BLOCK                                   (2<<6)
#define CASCADE                                 (3<<6)

#define ADDR_INC                                0
#define ADDR_DEC                                (1<<5)

#define AUTO_INIT_DISABLE                       (1<<4)
#define AUTO_INIT_ENABLE                        0

#define WRITE                                   (1<<2)
#define READ                                    (2<<2)

#define SELECT_CHAN(ch)          ((ch<4)       ?      ch : (ch-4))

#define ENABLE(ch)               ((ch<4)       ?      ch : (ch-4))
#define DISABLE(ch)              ((1<<2) | ((ch < 4) ? ch : (ch-4)))

#define INPUT                                   0
#define OUTPUT                                  1
```

G.2 IDList.h

```
#ifndef ID_LIST_H
#define ID_LIST_H

#include <arpa/inet.h>

/*************** ID LIST Definitions and macros ***************/
/* DO NOT EDIT */
typedef union {
char asChar[2];
uint16_t asUintN;
} ID_t;

typedef struct {
const char * ID_Name;
const unsigned int ID_Value;
} IDENTRY_t;

#define IDMAP_SIZE sizeof(IDMAP) / sizeof(IDENTRY_t)

#define BEGIN_IDMAP const IDENTRY_t IDMAP[]={
#define END_IDMAP {"",0}};
#define IDNAME2IDNR(x) x
#define IDENTRY(x) {(#x),IDNAME2IDNR(x)},


#define ID_TEXTMESSAGE     0  // Transmits a plain textmessage
#define ID_RQIDNAMES       1  // Request id names (Client -> Server)
#define ID_TXIDNAME        2  // IDName Package (Server -> Client)
#define ID_IDNAMECOMPLETE  3  // IDName transmission complete (Server ->
                              // Client)
#define ID_SENDDATA        4  // Used Internally by the Server to transfer
                              // message information
#define ID_USER 10 // The first legal User ID
/* END - DO NOT EDIT */
/*****************************************************************/



/************************* Type ID List ****************************/

// Always begin your ID's with the characters "ID_" and please use
// capital letters for conformance
#define ID_CHAN0 ID_USER+0 // Analog Channel 0
#define ID_CHAN1 ID_USER+1 // Analog Channel 1
#define ID_CHAN2 ID_USER+2 // Analog Channel 2
#define ID_CHAN3 ID_USER+3 // Analog Channel 3
#define ID_CHAN4 ID_USER+4 // Analog Channel 4
#define ID_CHAN5 ID_USER+5 // Analog Channel 5
#define ID_CHAN6 ID_USER+6 // Analog Channel 6
#define ID_CHAN7 ID_USER+7 // Analog Channel 7

// Client Commands:
#define ID_START_AD 1024 // Client/Server requests A/D conversion to start
#define ID_STOP_AD 1025 // Client/Server requests A/D conversion to halt
#define ID_SET_SERVO_0 1026 // Client requests servo 0 to move to a new position
#define ID_OPEN_VALVE_0 1027 // Opens a solenoid valve connected to an I/O-port
#define ID_CLOSE_VALVE_0 1028 // Close a solenoid valve connected to an I/O port

/* Always remember to update the IDMAP below to mirror all the changes made above
*/

BEGIN_IDMAP
IDENTRY(ID_CHAN0)
IDENTRY(ID_CHAN1)
IDENTRY(ID_CHAN2)
IDENTRY(ID_CHAN3)
```

```
IDENTRY(ID_CHAN4)
IDENTRY(ID_CHAN5)
IDENTRY(ID_CHAN6)
IDENTRY(ID_CHAN7)
IDENTRY(ID_START_AD)
IDENTRY(ID_STOP_AD)
IDENTRY(ID_SET_SERVO_0)
IDENTRY(ID_OPEN_VALVE_0)
IDENTRY(ID_CLOSE_VALVE_0)
END_IDMAP

#endif
```

G.3 ioports.h

```
/*Digital port control macros*/

#define PCM3718_BASE_ADRESS 0x210 //The base adress must correspond
                                  // to the physical adress of the card

#define PPORT 0x378               // Defualt Parallell port adress
#define IOPORTLOW PCM3718_BASE_ADRESS+3 // Low IO-port  (DAQ-card)
#define IOPORTHIGH PCM3718_BASE_ADRESS+11 // High IO-port (DAQ-card)
#define PIN1 1                          // port output pin 1
#define PIN2 2                          // port output pin 2
#define PIN3 4                          // port output pin 3
#define PIN4 8                          // port output pin 4
#define PIN5 16                         // port output pin 5
#define PIN6 32                         // port output pin 6
#define PIN7 64                         // port output pin 7
#define PIN8 128                        // port output pin 8

// Internal state variables
unsigned char _parportstate=0;
unsigned char _ioportlowstate=0;
unsigned char _ioporthighstate=0;

/* This macros can be used to control the parallell port. */
#define PPortOut(value) (out8(PPORT,_parportstate=value))
// Sets the entire Parallelll port

/* This Macro-function only sets the desired pin to on(=1) or off(=0) */
#define SetPPin(pin,value) (out8(PPORT,_parportstate=(value>0? _parportstate | pin
: _parportstate & ~pin)))

/* Similar function as above, but applies to the PCM3718H digital ports */
#define IOPortlow(value) (out8(IOPORTLOW,_ioportlowtstate=value))
#define IOPortHigh(value) (out8(IOPORTHIGH,_ioporthightstate=value))
#define SetPinLow(pin,value) (out8(IOPORTLOW,_ioportlowstate=(value>0?
_ioportlowstate | pin : _ioportlowstate & ~pin)))
#define SetPinHigh(pin,value) (out8(IOPORTHIGH,_ioporthighstate=(value>0?
_ioporthighstate | pin : _ioporthighstate & ~pin)))
```

G.4 pulsecodes.h

```
/****************** Pulse Codes *********************************/
// Pulses are sent to threads for synchronisation
// These are used by the PCM3718 driver as well.
// Please don't delete the ones you don't use.

#define TIMER_PULSE_CODE   _PULSE_CODE_MINAVAIL +0
// Used by various timers

#define RELEASE_PULSE_CODE   _PULSE_CODE_MINAVAIL+1
// Used to activate a non-active thread

#define SHUTDOWN_PULSE_CODE  _PULSE_CODE_MINAVAIL+2
// Send to a thread to safely shut it down

#define RESET_PULSE_CODE  _PULSE_CODE_MINAVAIL+3
// Resets the state of a running thread
// (usually to a non-active state)

#define FILTER_PULSE_CODE  _PULSE_CODE_MINAVAIL+4
// Used by the PCM3718H driver to inform the thread
// that filter data is ready for retreival

#define COMMAND_PULSE_CODE _PULSE_CODE_MINAVAIL + 5
// Sent by the Listen thread to other threads
// when relaying client commands.
```

G.5 servocontrol.h

```
/************ Servo control functions ******************/
#include <termios.h>

/* Opens the serial servo controller @ 9600 baud
* Input:
*      devicename - a string containing the device to open, for example *
               "/dev/ser1"
* Output:
*      A file descriptor to the servo controller or -1 on an error
*/

int servo_open(const char *devicename)
{
 int fd;
 struct termios tio;
 if ((fd=open(devicename,O_RDWR))<0) return (-1);
 cfmakeraw(&tio);
 tio.c_cflag &= ~(IXON|IXOFF);
 tio.c_cflag |= (CLOCAL | CREAD);
 tio.c_cflag &= ~PARENB;
 tio.c_cflag &= ~CSTOPB;
 tio.c_cflag &= ~CSIZE;
 tio.c_cflag |= CS8;
 cfsetispeed(&tio,B9600);
 cfsetospeed(&tio,B9600);
 tcsetattr(fd,TCSANOW,&tio);
 memset(&tio,0,sizeof(tio));
 tcgetattr(fd,&tio);
 if(cfgetispeed(&tio)!=B9600) return -1;
 return fd;
}

/* Sets the servo position of a servo
* Input:
*             fd - the filedescriptor obtained from servo_open
*             servo - An integer containing which servo to move (0-7)
*             pos - An integer containing the new servo position (0-254)
*/
void set_servo(int fd,unsigned char servo,unsigned char pos)
{
       int n;
       unsigned char buf[3]={0xFF,servo,pos};
       n = write(fd,buf,3);
       if (n<0) printf("write failure!\n");
}
```

G.6 pcm3718.h

```
#ifndef _PCM3718
#define _PCM3718
/***************** PCM-3718H Specific Macros *********************/

// W(rite): clear INT request
#define PCM3718_CLRINT 8
// R(ead): return status byte
#define PCM3718_STATUS 8
// R: A/D high byte W: A/D range control
#define PCM3718_RANGE 1
// R: next mux scan channel W: mux scan channel & range control pointer
#define PCM3718_MUX 2
// R/W: operation control register
#define PCM3718_CONTROL 9
// W: counter enable
#define PCM3718_PACERENABLE 10

// R: low byte of A/D W: soft A/D trigger
#define PCM3718_AD_LO 0
// R: high byte of A/D W: A/D range control
#define PCM3718_AD_HI 1
// R: low&high byte of DI
#define PCM3718_DIO_LO 3
#define PCM3718_DIO_HI 11
// W: low&high byte of DO
// counters
#define PCM3718_PACER0 12
#define PCM3718_PACER1 13
#define PCM3718_PACER2 14
// W: counter control
#define PCM3718_PACERCTL 15

// IRQ interrupt control values
#define USE_IRQ2 0xA0
#define USE_IRQ3 0xB0
#define USE_IRQ4 0xC0
#define USE_IRQ5 0xD0
#define USE_IRQ6 0xE0
#define USE_IRQ7 0xF0

// For Internal software use
#define IRQ2 2
#define IRQ3 3
#define IRQ4 4
#define IRQ5 5
#define IRQ6 6
#define IRQ7 7

// Card control modes
#define USE_PACER 3
#define USE_SOFT 0
#define USE_DMA 4

// Read/Write modes
#define RWMODE_DIO1 0
#define RWMODE_DIO2 1
#define RWMODE_ANALOG 2

// Name for shared memory area
#define SHM_NAME "/dev/shmem/pcm3718"

// Resource manager run modes
#define VERBOSE 2
#define NORMAL 1
#define SILENT 0
/* Types used when setting up the range of the A/D card */
typedef enum {BIPOLAR_5V,BIPOLAR_2_5V, BIPOLAR_1_25V,BIPOLAR_0_625V,
```

```
UNIPOLAR_10V, UNIPOLAR_5V, UNIPOLAR_2_5V, UNIPOLAR_1_25V, BIPOLAR_10V} Rangetype;

/* Struct filled by client when setting up the A/D card */
typedef struct _devctl_channelstruct {
unsigned char start,stop; //The A/D converter starts converting the
// "start" channel, then for each conversion.
// The converter automatically inrements the current channel by 1.
// After convering channel "stop", it automatically starts at channel
// "start" again.
Rangetype range[16]; //See Rangetype above. Note that only the channels
// between "start" and "stop" needs
// to be fiiled. 16 channels is the maximum number of available
// channels.
} channel_setup_data_t;

/* struct used to hold the two numbers used to set the
   internal pacer clock */
typedef struct {
unsigned int c1;
unsigned int c2;
} double_uint;

/* Struct filled by client when attaching a filter thread to this resource manager
*/
typedef struct {
int channelid; // Channel id of a channel created by the client process using
ChannelCreate()
pid_t pidid; // The Process ID of the calling thread/process.
} attachfiltersend_t;

/* Struct filled by resource manager when replying to a filter
attach operation*/
typedef struct {
unsigned int size; // The size in bytes of the shared memory area.
unsigned char shm_name[30]; // The name of the shared memory area (Same as
SHM_NAME, above)
} attachfilterreply_t;

/* Union containing both the message and reply structures of the attach filter
message passing */
typedef union {
attachfiltersend_t data;
attachfilterreply_t reply;
} attach_message_t;

/******** This is the Devctl Control Macros *****************/
#define CMD_CODE 1 // Just an arbitrary number used as a relative
                   // reference for the manager.

#define DEVCTL_SETCHANNEL __DIOTF(_DCMD_MISC, CMD_CODE + 0,channel_setup_data_t) //
Sets the channel range of the card
#define DEVCTL_SETPACER __DIOTF(_DCMD_MISC, CMD_CODE + 1,double_uint) // Set the
pacer speed
#define DEVCTL_DMA_INIT __DIOTF(_DCMD_MISC, CMD_CODE + 2,NULL)
// Inits DMA-controller and buffers
#define DEVCTL_AD_START __DIOTF(_DCMD_MISC, CMD_CODE + 3,NULL)
// Start A/D conversion
#define DEVCTL_AD_STOP __DIOTF(_DCMD_MISC, CMD_CODE + 4,NULL)
// Stops A/D conversion
#define DEVCTL_ATTACHFILTER __DIOTF(_DCMD_MISC, CMD_CODE + 5,attach_message_t) //
Attaches a filter for synchronization
#define DEVCTL_SET_READMODE __DIOTF(_DCMD_MISC, CMD_CODE + 6,unsigned char) // Sets
the mode of the read/write operations
#define DEVCTL_SET_TRANSFER_SPEED __DIOTF(_DCMD_MISC, CMD_CODE + 7,unsigned int) //
Sets the filter wake-up speed
/*********************************************************************/

/* Internal structure used by Resource manager to manage the status of the card. */
typedef struct {
int iobase; // Base adress of device
unsigned char *dmabuf,*shmem_area; // DMA-buffer and shared memory area.
```

G.6 **pcm3718.h**

```
unsigned int buf1size,shmem_size; // DMA- and shared memory buffer sizes.
Rangetype range[16];              // The Conversion ranges for each channel
unsigned char start_channel; // The Conversion start channel
unsigned char stop_channel; // Conversion Stop Channel
unsigned char dma_channel; // DMA-channel used
unsigned int CLK_speed;   // External Clock speed
unsigned int C1;         // First pacer control value
unsigned int C2;        // Second pacer control value
unsigned char IRQ;     // IRQ used
unsigned int transfer_rate;        // Filter wake-up speed in Hz
unsigned int transfer_timeperiod; // Equivalent speed in nanoseconds.
} PCM3718_PRIVATE;

#endif
```

G.7 pcm3718.c

```
/****************PCM3718H Resource Manager ********************/

#include <stdio.h>
#include <hw/inout.h>
#include <sys/mman.h>
#include <sys/neutrino.h>
#include <inttypes.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <sys/netmgr.h>
#include <sched.h>
#include <time.h>
#include <unistd.h>
#include <errno.h>
#include <stddef.h>
#include <stdlib.h>
#include <devctl.h>
#include <sys/iofunc.h>
#include <sys/dispatch.h>
#include "dmadef.h"
#include "pcm3718.h"
#include "pulsecodes.h"


/***************** Resource manager variables ****************/
static resmgr_connect_funcs_t connect_funcs;
static resmgr_io_funcs_t io_funcs;
static iofunc_attr_t attr;
/*************************************************************/

/********* Global Variables ***********/
int runmode;
// The resource manager can be executed in 3 modes:
// SILENT : No messages are written to screen.
// NORMAL: Only some messages (i.e. errors)~are written to screen.
// VERBOSE: Prints all current operations to screen, including errors.

// The channel id to the copy thread (see Copy_thread())
int channelid;
// A Structure containing all information about the device
PCM3718_PRIVATE dev;
// Structs for changing the realtime clock
struct _clockperiod clockp,oldclock;
struct sigevent intevent;
// Predefined Interrupt event, used with the DMA-buffers
int chid, int_id; // Internal variables
int rwmode=RWMODE_DIO1; // Current read/write mode

/* A bitfieild vector containing the state of the Resource manager.
Prevents errors like trying to start A/D conversion before setting up the card
properly */

struct
{
unsigned device_open: 1;
unsigned dma_set : 1;
unsigned channel_set : 1;
unsigned pacer_set : 1;
unsigned ad_running : 1;
unsigned filter_attached : 1;
} status;


/* A message structure. Contains all potential messages used by the program */
typedef union {
```

```
struct _pulse pulse;
attachfiltersend_t channelinfo;
/* your other message structures would go here too */
} my_message_t;


const struct sigevent *isr_handler (void *arg, int id);


/*
============================================================
Detects the PCM-3718 cards
returns:
0 on success, 1 on failure
*/

static int PCM3718_detect(int iobase)
{
struct timespec ts;
ts.tv_sec=0;
ts.tv_nsec=2000000; //2 milliseconds

out8(iobase + PCM3718_MUX,0x00);
nanosleep(&ts,NULL);
if (in8(iobase + PCM3718_MUX)!=0x00) return 1;//there's no card
out8(iobase + PCM3718_MUX,0x55);
nanosleep(&ts,NULL);
if (in8(iobase + PCM3718_MUX)!=0x55) return 1;//there's no card
out8(iobase + PCM3718_MUX,0x00);
nanosleep(&ts,NULL);
out8(iobase + PCM3718_CONTROL,0x18);
nanosleep(&ts,NULL);
if (in8(iobase + PCM3718_CONTROL)!=0x18) return 1;

return 0; // ok, card exist
}


/*
============================================================
reset whole PCM-3718 cards

Input:
Most of the functions with names beginning with "PCM3718_" get their indata from
the
global structure named "dev".
*/
static void PCM3718_reset()
{
struct timespec ts;
ts.tv_sec=0;
ts.tv_nsec=2000000; //2 milliseconds

out8(dev.iobase + PCM3718_DIO_HI,0);
out8(dev.iobase + PCM3718_DIO_LO,0);
nanosleep(&ts,NULL);
out8(dev.iobase + PCM3718_CONTROL,0);
out8(dev.iobase + PCM3718_PACERENABLE,0xff);
out8(dev.iobase + PCM3718_MUX,0);
out8(dev.iobase + PCM3718_CLRINT,0);
out8(dev.iobase + PCM3718_PACERCTL,0xb0);/* Stop pacer */
out8(dev.iobase + PCM3718_PACERCTL,0x70);
out8(dev.iobase + PCM3718_PACERCTL,0x30);
out8(dev.iobase + PCM3718_RANGE,0);
}

/*
============================================================
Helper function for setrange */

Rangetype2str(Rangetype range,char *str)
```

```
{
switch(range) {
case BIPOLAR_5V: strcpy(str,"BIPOLAR 5V"); break;
case BIPOLAR_2_5V: strcpy(str,"BIPOLAR 2.5V"); break;
case BIPOLAR_1_25V: strcpy(str,"BIPOLAR 1.25V"); break;
case BIPOLAR_0_625V: strcpy(str,"BIPOLAR 0.625V"); break;
case UNIPOLAR_10V: strcpy(str,"UNIPOLAR 10V"); break;
case UNIPOLAR_5V: strcpy(str,"UNIPOLAR 5V"); break;
case UNIPOLAR_2_5V: strcpy(str,"UNIPOLAR 2.5V"); break;
case UNIPOLAR_1_25V: strcpy(str,"UNIPOLAR 1.25V"); break;
case BIPOLAR_10V: strcpy(str,"BIPOLAR 10V");
}
}

/*
============================================================
Set Input range of card
*/

PCM3718_setrange()
{
int n;
struct timespec ts;
int se;
char buffer[20];
se=(in8(dev.iobase+PCM3718_STATUS)&0x20)? 1 : 0;
ts.tv_sec=0;
ts.tv_nsec=1000000;
// Check for range consistency
if (dev.start_channel>dev.stop_channel) return EINVAL;
if (se && dev.stop_channel>15) return EINVAL;
else if (dev.stop_channel>7) return EINVAL;

// If range is ok, start Setting up the card using the Rangetype
// array
for (n=dev.start_channel; n<=dev.stop_channel;++n)
{
if ((int) (dev.range[n]) > 8) return EINVAL;
// There are 9 different configurations supported by the card
out8(dev.iobase+PCM3718_MUX,n);
if (runmode==VERBOSE) {
Rangetype2str(dev.range[n],buffer);
printf("PCM3718: Setting channel %d to mode %s\n",n,buffer);}
out8(dev.iobase+PCM3718_RANGE,(int) (dev.range[n]));
}
nanosleep(&ts,NULL);
out8(dev.iobase+PCM3718_MUX,(dev.stop_channel<<4) | dev.start_channel);
n=in8(dev.iobase+PCM3718_MUX);
if (se && runmode!=SILENT) printf("PCM3718: A/D-range between %d…
and %d has been set up for SE mode\n",n&0xf,n>>4);
else if (runmode!=SILENT) printf("PCM3718: A/D-range between %d…
and %d has been set up for DIFF mode\n",n&0xf,n>>4);
return 0;
}

/*
============================================================
Software triggered A/D function
*/
```

G.7 pcm3718.c

```
int PCM3718_software_AI()
{
unsigned int reading;
struct timespec ts;
ts.tv_sec=0;
ts.tv_nsec=1000000; //1 millisecond
reading=0;
out8(dev.iobase+PCM3718_STATUS,0);
out8(dev.iobase+PCM3718_AD_LO,0); // write something to base+0
while (1)
{
nanosleep(&ts,NULL);
if (in8(dev.iobase+PCM3718_STATUS)&20) break;
nanosleep(&ts,NULL);
}
reading= in16(dev.iobase+PCM3718_AD_LO);
// Note: This only works on
// Intel Architecture machines (little-endian)
return reading;
}

/*
============================================================
Reads Digital IO
returns:
A byte containing entire channel (8 bits)
*/

unsigned char PCM3718_DIO_read( )
{
unsigned char data;
if (rwmode==RWMODE_DIO1) data=in8(dev.iobase+PCM3718_DIO_LO);
else data=in8(dev.iobase+PCM3718_DIO_HI);
return data;
}


/*
============================================================
Writes to the Digital IO
Input:
A byte (8 bits) to write to the channel
*/
void PCM3718_DIO_write(unsigned int data)
{
if (rwmode==RWMODE_DIO1) out8(dev.iobase+PCM3718_DIO_LO,(unsigned char) data);
else out8(dev.iobase+PCM3718_DIO_HI,data);
}



/*
============================================================
Sets the pacer speed with the designated values C1 and C2 */


PCM3718_StartPacer(unsigned int C1,unsigned int C2)
{
out8(dev.iobase+PCM3718_PACERCTL,0x76);
out8(dev.iobase+PCM3718_PACER1,C1 & 0xff);
out8(dev.iobase+PCM3718_PACER1,C1 >> 8);
out8(dev.iobase+PCM3718_PACERCTL,0xB6);
out8(dev.iobase+PCM3718_PACER2,C2 &0xff);
out8(dev.iobase+PCM3718_PACER2,C2 >> 8);
}


/*
============================================================
Initailizes the DMA Buffers */
```

G.7 pcm3718.c

```
int PCM3718_setupBuffers()
{
int speed,fd;

// These three lines calculates how many shared memory buffers
// you can fit into a block of 65536 bytes
speed=dev.CLK_speed/(double) (dev.C1*dev.C2);
dev.shmem_size=(int) speed/dev.transfer_rate*2;
dev.buf1size=65536-(65536 % dev.shmem_size);

// Start allocating memory
if((dev.dmabuf = mmap( 0,
dev.buf1size,PROT_READ|PROT_WRITE|PROT_NOCACHE,
// Allow the Kernel to map memoryspace which is accessable by the
// DMA Controller
MAP_PRIVATE|MAP_PHYS|MAP_BELOW16M |MAP_ANON|MAP_NOX64K,
NOFD,0 ) ) == MAP_FAILED)
{
if (runmode!=SILENT) perror("PCM3718");
abort();
}


/* DMA-buffer allocation OK, now create the shared memory area */

shm_unlink(SHM_NAME); // Unlinks any previously linked memory areas

if ((fd=shm_open(SHM_NAME, O_CREAT | O_RDWR | O_EXCL, 0770)) == -1)
{
if (runmode!=SILENT) perror("PCM3718");
close(fd);
abort();
}

// Expand the memory area to desired size
if ( shm_ctl(fd, SHMCTL_GLOBAL | SHMCTL_ANON, 0, dev.shmem_size) == -1)
{
if (runmode!=SILENT) perror("PCM3718");
close(fd);
shm_unlink(SHM_NAME);
abort();
}
// map the shared memory area
if( (dev.shmem_area = mmap(NULL, dev.shmem_size,
PROT_READ | PROT_WRITE | PROT_NOCACHE,
MAP_SHARED, fd, 0)
) == MAP_FAILED)
{
if (runmode!=SILENT) perror("PCM3718");
close(fd);
shm_unlink(SHM_NAME);
abort();
}
close(fd);

// clear the shared memory area
memset(dev.shmem_area,0,dev.shmem_size);
// the two memory buffers are now ready to be used.
return 0;
}
```

G.7 pcm3718.c

```
/*
========================================================================
Initializes the DMA controller
Input:
dmaChan - the Dma channel to be used
startAdress - A memory buffer to store the incoming data
transferCount - Number of bytes to transfer before resetting the
buffer
autoInitMode - A boolean (0/1) telling if the Controller
should reinitialize the buffer after transfer
completion
dir - Tells the dma-controller if we want to do reading or
writing to/from the buffer
*/

int DMASetup (int dmaChan, void *startAddress, unsigned int transferCount,
unsigned int autoInitMode,unsigned int dir)
{

/* Note: This function uses some fancy low-level programming
techniques which I don't have time to explain.
*/

unsigned char lsb,msb,page,lcnt,hcnt,dmaMode;
off_t addr; // Physical Memory location (also called memory
             // offset) of StartAdress
unsigned int maxTransferCnt;
if(mem_offset(startAddress, NOFD, 1, &addr, 0) == -1) return -1;
// Memory offset fault
maxTransferCnt = 0x10000 - ((dmaChan < 4 ? addr : addr >>1) &
0xffff);
if (transferCount > maxTransferCnt) return -1;
if (dmaChan < 4)
{
lsb =addr;
msb = addr>> 8;
page = addr >>16;
}
else
{
lsb = addr>>1;
msb = addr>> 9;
page = addr >>16 & 0xfe;
}
--transferCount;
lcnt = transferCount;
hcnt = transferCount >> 8;

dmaMode = SINGLE | ADDR_INC | (autoInitMode ? AUTO_INIT_ENABLE :
AUTO_INIT_DISABLE) | (dir== INPUT ? WRITE : READ) |
SELECT_CHAN(dmaChan);
InterruptDisable();
out8(SNGL_MASK_reg(dmaChan) , DISABLE(dmaChan));
out8(MODE_reg(dmaChan) , dmaMode);
out8(CLR_BYTE_PTR_reg(dmaChan), 0);
out8(BASE_ADDR_reg(dmaChan) , lsb);
out8(BASE_ADDR_reg(dmaChan) , msb);
out8(PAGE_ADDR_reg(dmaChan) , page);
out8(CLR_BYTE_PTR_reg(dmaChan), 0);
out8(COUNT_reg(dmaChan) , lcnt);
out8(COUNT_reg(dmaChan) , hcnt);
InterruptEnable();
return 0;
}
```

G.7 pcm3718.c

```
/****************** Helper-thread implementation ******************/
/* This thread automatically copies the most recent data from the DMA-buffer into
an inter-process (QNX) shared memory area.
The thread also sends out a "wake-up" signal to the filter-process telling new data
is available, if a filter is attached to this thread, that is. */

void * copy_thread (void * devicestruct)
{
struct sigevent timerevent;
unsigned char * maxbuf;
unsigned char *dmaptr;
unsigned char filterattached;
struct itimerspec itime;
timer_t timer_id;
int rcvid,chid2,status,policy; // chid2 is the channel id to
                               // communicate with the filter,
                               // chid is for in process comm.
struct sched_param param;
my_message_t msg;

dmaptr=dev.dmabuf;
filterattached=0;
chid=chid2=-1;
maxbuf=dev.dmabuf+dev.buf1size - dev.shmem_size;
int_id=InterruptAttach(IRQ5, isr_handler,NULL,0,0);

// boost this thread's priority
pthread_getschedparam(pthread_self(),&policy,&param);
param.sched_priority=
sched_get_priority_max(sched_getscheduler(0))-1;
pthread_setschedparam(pthread_self(),policy,&param);
// create a communication channel to be used by the main process
chid = ChannelCreate(0);

/****************** Create a timer ************************/
timerevent.sigev_notify = SIGEV_PULSE;
timerevent.sigev_coid = ConnectAttach(ND_LOCAL_NODE,
0,chid,_NTO_SIDE_CHANNEL, 0);
timerevent.sigev_priority =
sched_get_priority_max(sched_getscheduler(0));
timerevent.sigev_code = TIMER_PULSE_CODE;
timer_create(CLOCK_REALTIME, &timerevent, &timer_id);

dev.transfer_timeperiod= 1000000000 / dev.transfer_rate;
itime.it_value.tv_sec = 0;
itime.it_value.tv_nsec = dev.transfer_timeperiod;
itime.it_interval.tv_sec = 0;
itime.it_interval.tv_nsec = dev.transfer_timeperiod;
/*************************************************************/
if (runmode==VERBOSE)
printf("PCM3718: Copy-thread ready and waiting\n");


/* Main message receive loop */
for (;;)
{
rcvid = MsgReceive(chid, &msg, sizeof(msg), NULL);
if (rcvid == 0)
{ /* we got a pulse */
if (msg.pulse.code == TIMER_PULSE_CODE)
// We got a timer pulse, time to copy the
// most recent data out of the buffer
{
{
memcpy(dev.shmem_area,dmaptr,dev.shmem_size);
if (filterattached)
// Send a pulse to the process containing a filter thread
MsgSendPulse(chid2,
sched_get_priority_max(sched_getscheduler(0)),
FILTER_PULSE_CODE,0);
dmaptr+=dev.shmem_size;
```

```
if (dmaptr==maxbuf)
// If we are at the end of the buffer,
// we must wait for a interrupt signal
// To synchronize the timer.
{
//Then copy the last data chunk from
//the DMA-buffer before resetting the timer.
InterruptWait(NULL,NULL);
memcpy(dev.shmem_area,dmaptr,dev.shmem_size);
dmaptr=dev.dmabuf;
InterruptUnmask(IRQ5,-1);
if (filterattached)
MsgSendPulse (chid2,sched_get_priority_max(sched_getscheduler(0)),
FILTER_PULSE_CODE,0);
timer_settime(timer_id, 0, &itime, NULL);
// The timer signal is supposed to be a bit behind the DMA-transfers
// to guarantee that the transfer really has finished before copying.
}
}
} /* else other pulses ... */
else if (msg.pulse.code == RELEASE_PULSE_CODE)
// This activates the copy thread
{
timer_settime(timer_id, 0, &itime, NULL);
}
else if (msg.pulse.code == RESET_PULSE_CODE)
{
timer_delete(timer_id);
timer_create(CLOCK_REALTIME, &timerevent, &timer_id);
}
else if (msg.pulse.code == SHUTDOWN_PULSE_CODE)
{
break; //jump out of loop
}
}
else
{ /* else other messages ... */
// There is only one type of message that can be received by this
// thread.

// Connect to the filters message channel
chid2=ConnectAttach(ND_LOCAL_NODE, msg.channelinfo.
pidid,msg.channelinfo.channelid,_NTO_SIDE_CHANNEL, 0);
if (chid2==-1) // Cannot open channel;
{
if (runmode!=SILENT)
fprintf(stderr,"PCM3718: Error attaching channel!\n");
status=1;
}
else {status=0; filterattached=1; }// Filter is now attached.
MsgReply(rcvid,0,&status,sizeof(status));
}
}
if (runmode==VERBOSE)
printf("PCM3718: Shutting down Copy thread...\n");
InterruptDetach (int_id);
timer_delete(timer_id);
if (chid!=-1) ChannelDestroy(chid);
if (chid2!=-1) ConnectDetach(chid2);
}

// this is the ISR
const struct sigevent *
isr_handler (void *arg,int id)
{
// Mask the interrupt (to prevent the kernel from getting the same interrupt again)
InterruptMask(IRQ5,-1);
out8(dev.iobase+PCM3718_CLRINT,0); // Clear Interrupt
return (&intevent);
}
```

G.7 pcm3718.c

```
/*******************Device Manager functions********************/

int handle_devctl(resmgr_context_t *ctp, io_devctl_t *msg,
RESMGR_OCB_T *ocb);

int io_devctl(resmgr_context_t *ctp, io_devctl_t *msg,
RESMGR_OCB_T *ocb)
{
int nbytes, ret, previous,retstatus,chid2,USE_IRQN;
double speed;
union {
channel_setup_data_t chdata;
double_uint pacerdata;
attach_message_t amdata;
unsigned char charvalue;
unsigned int uintvalue;
// ... other devctl types you can receive
} *rx_data;

/*
Let common code handle DCMD_ALL_* cases.
You can do this before or after you intercept devctl's depending
on your intentions.
*/
if ((retstatus = iofunc_devctl_default(ctp, msg, ocb)) != _RESMGR_DEFAULT) {
return(retstatus);
}
retstatus = nbytes = 0;

/*
Note: this assumes that you can fit the entire data portion of
the devctl into one message. In reality you should probably
perform a MsgReadv() once you know the type of message you
have received to suck all of the data in rather than assuming
it all fits in the message. We have set in our main routine
that we'll accept a total message size of up to 2k so we
don't worry about it.
*/
rx_data = _DEVCTL_DATA(msg->i);

/* Now check for all user defined messages */

/************************* DEVCTL MESSAGE HANDLER ***************/
switch (msg->i.dcmd) {

/*The printf() commands should keep you updated on what's going on */

case DEVCTL_SETCHANNEL: // Sets the channelmodes on each channel
if (runmode==VERBOSE)
printf("PCM3718: Setting up channels %d through %d...\n",
rx_data->chdata.start,rx_data->chdata.stop);
dev.start_channel=rx_data->chdata.start;
dev.stop_channel=rx_data->chdata.stop;
memcpy(dev.range,rx_data->chdata.range,sizeof(dev.range));
ret=PCM3718_setrange();
if (ret==EINVAL) { if (runmode==VERBOSE)
printf("Channel setup failed!\n"); return EINVAL; }
status.channel_set=1;
break;


case DEVCTL_SETPACER:
// Sets the pacer values C1 and C2 to be used in pacer mode
if (runmode==VERBOSE)
printf("PCM3718: Setting up Pacer values to %u and %u...\n",
rx_data->pacerdata.c1,rx_data->pacerdata.c2);
// Check for errors
if (status.dma_set)
{
if (runmode!=SILENT) fprintf(stderr,
"PCM3718: Cannot set clock speed after DMA buffers have been …
```

```
created\n\t close connection before calling SETPACER again\n");
return EINVAL;
}

if (rx_data->pacerdata.c1<2 || rx_data->pacerdata.c2<2) {
if (runmode==VERBOSE)
fprintf(stderr,"PCM3718: Unallowed Pacervalue entered\n");
return EINVAL; }
dev.C1=rx_data->pacerdata.c1; dev.C2=rx_data->pacerdata.c2;
status.pacer_set=1;
break;

case DEVCTL_DMA_INIT:
// Initializes the DMA-buffers and sets up the DMA-controller
if (dev.dmabuf!=NULL || dev.shmem_area!=NULL || status.dma_set)
{
if (runmode!=SILENT)
fprintf(stderr,"PCM3718: DMA buffers already exists!\n\t …
close connection before calling START_DMA again\n");
return EINVAL;
}
if (runmode==VERBOSE)
printf("PCM3718: Setting up DMA-buffers...\n");
if (PCM3718_setupBuffers(&dev)) return EINVAL;
if (runmode==VERBOSE)
printf("PCM3718: Starting DMA-controller on channel …
%d...\n",dev.dma_channel);
if(DMASetup(dev.dma_channel,dev.dmabuf,dev.buf1size,0,INPUT)!=0) {
if (runmode!=SILENT)
fprintf(stderr,"PCM3718: DMA setup error!\n"); return EIO; }
if (runmode==VERBOSE)
fprintf(stderr,"PCM3718: Starting up DMA copy-thread...\n");
pthread_create (NULL, NULL, copy_thread,NULL);
speed=dev.CLK_speed/ (double) (dev.C1*dev.C2);
if (runmode!=SILENT)
printf("PCM3718: A/D Conversion setup at %7.3f Hz\n …
(Generating %7.3f bytes of data)\n",speed,2*speed);
if (runmode==VERBOSE)
printf("PCM3718: Total DMA-buffer: %d bytes\n",dev.buf1size);
// Create a comm. link to copy-thread.
channelid=ConnectAttach(ND_LOCAL_NODE, 0,chid,_NTO_SIDE_CHANNEL, 0);
status.dma_set=1;
break;
case DEVCTL_AD_START:
if (!(status.channel_set && status.pacer_set && status.dma_set &&
!status.ad_running))
{
if (runmode==VERBOSE)
{
fprintf(stderr,"PCM3718: Unable to start AD-conversion\n\t ");
fprintf(stderr,"channel_set:%u\n\t pacer_set:%u\n\t …
dma_set:%u\n\t ad_running:%u\n", status.channel_set,
status.pacer_set,status.dma_set,status.ad_running);
}
return EINVAL;
}
if (runmode==VERBOSE)
printf("PCM3718: Starting A/D conversion\n");
switch (dev.IRQ)
{
case IRQ2:
USE_IRQN=USE_IRQ2; break;
case IRQ3:
USE_IRQN=USE_IRQ3; break;
case IRQ4:
USE_IRQN=USE_IRQ4; break;
case IRQ5:
USE_IRQN=USE_IRQ5; break;
case IRQ6:
USE_IRQN=USE_IRQ6; break;
case IRQ7:
```

```
USE_IRQN=USE_IRQ7; break;
}
// If the current clock period is less than required by the filter,
// adjust it to the required speed;
ClockPeriod(CLOCK_REALTIME,NULL,&clockp,0);
if (clockp.nsec>dev.transfer_timeperiod)
{
if (runmode==VERBOSE)
printf("PCM3718: Updating Realtime-clock period\n");
clockp.nsec=dev.transfer_timeperiod;
clockp.fract=0;
ret=ClockPeriod(CLOCK_REALTIME,&clockp,&oldclock,0);
if (ret==-1) {if (runmode!=SILENT)
printf("Error setting clockperiod!\n"); return EIO;}
}
// Enable DMA-channel
out8(SNGL_MASK_reg(dev.dma_channel),ENABLE(dev.dma_channel));
// Setup PCM3718 card
out8(dev.iobase+PCM3718_CONTROL, USE_IRQN | USE_PACER | USE_DMA);
PCM3718_StartPacer(dev.C1,dev.C2); // Start the pacer
// Start the copy...
MsgSendPulse (channelid,sched_get_priority_max(sched_getscheduler(0)),
RELEASE_PULSE_CODE,0);
status.ad_running=1; // Update status vector
break;
case DEVCTL_AD_STOP: // Temporarily halts A/D conversion.
if (!status.ad_running)
{
if (runmode!=SILENT)
fprintf(stderr,"PCM3718: cannot stop ad-conversion since it's not …
running!\n");
return EINVAL;
}
if (runmode==VERBOSE) printf("PCM3718: Stopping A/D conversion\n");
out8(dev.iobase+PCM3718_CONTROL,USE_SOFT); // Disable PACER & DMA & IRQ
out8(SNGL_MASK_reg(dev.dma_channel),DISABLE(dev.dma_channel));
MsgSendPulse (channelid,sched_get_priority_max(sched_getscheduler(0)),
RESET_PULSE_CODE,0);
if (oldclock.nsec!=0) //AD_START has been called.
{
ret=ClockPeriod(CLOCK_REALTIME,&oldclock,NULL,0);
if (ret==-1) printf("PCM3718: Error resetting clockperiod!\n");
}
out8(dev.iobase + PCM3718_PACERCTL,0xb0); /* Stop pacer */
out8(dev.iobase + PCM3718_PACERCTL,0x70);
/* Stop pacer */
out8(dev.iobase + PCM3718_PACERCTL,0x30);
/* Stop pacer */
status.ad_running=0;
break;

case DEVCTL_ATTACHFILTER:
// Tells the Copy thread which process to wake up when data is ready
if (runmode==VERBOSE) printf("PCM3718: Attaching filter with chid %d and pid %d\n",
rx_data->amdata.data.channelid,rx_data->amdata.data.pidid);
MsgSend(channelid, &(rx_data->amdata.data), sizeof(attachfiltersend_t), &retstatus,
sizeof(status));
if (!retstatus)
{
strcpy(rx_data->amdata.reply.shm_name,SHM_NAME);
rx_data->amdata.reply.size=dev.shmem_size;
nbytes=sizeof(attachfilterreply_t);
}
else return EINVAL;
status.filter_attached=1;
if (runmode!=SILENT)
printf("PCM3718: Copy-thread ready to send %d bytes of data every %d…
ns\n",

dev.shmem_size,dev.transfer_timeperiod);
break;
```

```
case DEVCTL_SET_READMODE:
// Sets where the card will read from when issuing a (read)-command
if (rx_data->charvalue>RWMODE_ANALOG)
{
if (runmode!=SILENT) fprintf(stderr,"PCM3718: Cannot set Readmode\n");
return EINVAL;
}
rwmode=rx_data->charvalue;
break;

case DEVCTL_SET_TRANSFER_SPEED:
// Software mode to set the filter wakeup speed (also called transfer // rate)

// Check for errors
if (status.dma_set)
{
if (runmode!=SILENT)
fprintf(stderr,"PCM3718: You must set this speed before allocating … DMA-buffers.
Please Close connection before trying again\n");
return EINVAL;
}
if (runmode==VERBOSE)
printf("PCM3718: Setting filter wakeup speed to %d Hz\n",
rx_data->uintvalue);
dev.transfer_rate=rx_data->uintvalue;
break;
default:
return(ENOSYS);
}
/*********************************************************************/

/* Clear the return message ... note we saved our data _after_ this */
memset(&msg->o, 0, sizeof(msg->o));

/*
If you wanted to pass something different to the return
field of the devctl() you could do it through this member.
*/
msg->o.ret_val = retstatus;

/* Indicate the number of bytes and return the message */
msg->o.nbytes = nbytes;
return(_RESMGR_PTR(ctp, &msg->o, sizeof(msg->o) + nbytes));
}

/*=====================================================================
The io_open function handles open calls to this device.
It resets the entire card on success.
Hence, no more than one connection at a time should be open.
*/
int io_open (resmgr_context_t *ctp, io_open_t *msg,
RESMGR_HANDLE_T *handle, void *extra)
{
int ret;
if(status.device_open) // Allow only one open connection to this device
{
if (runmode!=SILENT) printf("PCM3718: This device is already in … use\n");
return EXIT_FAILURE;
}
// First run the default open manager....
ret=iofunc_open_default (ctp, msg, handle, extra);
if (ret!=EOK) return ret;
if (runmode==VERBOSE) printf("PCM3718: Device opened\n");
status.device_open=1;

// ....Then perform initializations, etc.
SIGEV_INTR_INIT(&intevent);
// Initialize an interrupt event. Used for interrupt handling.
PCM3718_reset(&dev);
```

```
if (runmode==VERBOSE) printf("PCM3718: Card has been reset\n");

return ret;
}

/*=====================================================================
The io_close function handles close calls to this device.
The QNX OS automatically calls this function when programs closes,
crashes or whatever. So there is no worry about memory leaks.
It mainly releases all memory allocated by SetupBuffer()-function and also resets
all
parameters.
*/


int
io_close(resmgr_context_t *ctp, void *msg, RESMGR_OCB_T *ocb)
{
int base,dma,clk,ret,irq,trate;
if (runmode==VERBOSE) printf("PCM3718: Closing connection...\n");
MsgSendPulse (channelid,sched_get_priority_max(sched_getscheduler(0)),
SHUTDOWN_PULSE_CODE,0);
out8(dev.iobase+PCM3718_CONTROL,USE_SOFT); // Disable PACER & DMA & IRQ
if (dev.dmabuf!=NULL) // DMA-Buffers are probably in use
{
out8(SNGL_MASK_reg(dev.dma_channel),DISABLE(dev.dma_channel));
if (runmode!=SILENT) printf("PCM3718: Releasing DMA-memory...\n");
if (munmap(dev.dmabuf,dev.buf1size)==-1 && runmode!=SILENT)
fprintf(stderr,"PCM3718: Error releasing mapped memory!\n");
if (munmap(dev.shmem_area,dev.shmem_size)==-1 && runmode!=SILENT)
fprintf(stderr,"PCM3718: Error releasing shared memory!\n");
if (shm_unlink(SHM_NAME) == -1 && runmode!=SILENT)
fprintf(stderr,"PCM3718: Error unlinking shared memory!\n");
}
if (oldclock.nsec!=0) //START AD has been called.
{
ret=ClockPeriod(CLOCK_REALTIME,&oldclock,NULL,0);
if (ret==-1 && runmode!=SILENT) printf("Error resetting clockperiod!\n");
}
/******** RESET ALL DEVICE PARAMETERS TO DEFAULT VALUES *************/
base=dev.iobase;
dma=dev.dma_channel;
clk=dev.CLK_speed;
irq=dev.IRQ;
trate=dev.transfer_rate;
memset(&dev,0,sizeof(dev)); // Reset Entire Device structure
// Reload constant parameters
dev.iobase=base;
dev.dma_channel=dma;
dev.CLK_speed=clk;
dev.IRQ=irq;
dev.transfer_rate=trate;
rwmode=RWMODE_DIO1;
if (runmode!=SILENT) printf("PCM3718: Connection closed\n");
// Reset the state vector
status.device_open=0;
status.dma_set=0;
status.channel_set =0;
status.pacer_set =0;
status.ad_running=0;
status.filter_attached=0;
return (iofunc_close_ocb_default(ctp, msg, ocb));
}

/*=========================================================
The io_read function handles read-function calls to this device.
It can perform 3 different operations depending on the Read/Write mode of this
device.
The Read/Write mode is set by issuing a devctl call with a SET_READMODE argument.
When readmode is:
RWMODE_DIO1- It reads the lower 8 bits of the Digital IO channel.
```

```
RWMODE_DIO2 - It read the higher 8 bits of the Digital IO channel.
RWMODE_ANALOG - It issues a manual A/D conversion. (Software mode)
*/
io_read (resmgr_context_t *ctp, io_read_t *msg, RESMGR_OCB_T *ocb)
{
int status;
unsigned int data;

if ((status = iofunc_read_verify (ctp, msg, ocb, NULL)) != EOK)
return (status);

if (msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE)
return (ENOSYS);

if (rwmode==RWMODE_DIO1 || rwmode==RWMODE_DIO2)
{
if (runmode==VERBOSE)
printf("PCM3718: reading from Digital channel\n");
data=PCM3718_DIO_read();
}
else if (rwmode==RWMODE_ANALOG)
{
if (runmode==VERBOSE) printf("PCM3718: reading from analog channel\n");
data=PCM3718_software_AI();
}
MsgReply(ctp->rcvid,EOK,&data,sizeof(data));
ocb->attr->flags |= IOFUNC_ATTR_ATIME;
return (EOK);
}

/*============================================================
The io_write function writes to the Digital IO channel.
Which part of the channel (lower or higher) it writes to, depends on
the value of rwmode (see above).
*/
int io_write (resmgr_context_t *ctp, io_write_t *msg, RESMGR_OCB_T *ocb)
{
int status;
unsigned int data;
char *buf;

if ((status = iofunc_write_verify(ctp, msg, ocb, NULL)) != EOK)
{
if (runmode==VERBOSE) printf("No write permission!\n");
return (status);
}

if (msg->i.xtype & _IO_XTYPE_MASK != _IO_XTYPE_NONE)
return(ENOSYS);

resmgr_msgread(ctp, &data, msg->i.nbytes, sizeof(msg->i));
if (runmode==VERBOSE) printf("PCM3718: Writiing to digital channel\n");
PCM3718_DIO_write(data);
return (EOK);
}

/*============================================================
The PCM3718 main function, starts and registers the /dev/pcm3718 device.
*/

main(int argc, char **argv)
{
/* declare variables we'll be using */
resmgr_attr_t resmgr_attr;
dispatch_t *dpp;
dispatch_context_t *ctp;
int id,i,fd;
char* check;
resmgr_connect_funcs_t connect_funcs;
resmgr_io_funcs_t io_funcs;
iofunc_attr_t attr;
```

```
ThreadCtl(_NTO_TCTL_IO, 0);
runmode=NORMAL;

/************ initalize default parameter ***************/
memset(&dev,0,sizeof(dev));
dev.iobase=0x210;
dev.dma_channel=3;
dev.CLK_speed=10000000;
dev.IRQ=IRQ5;
dev.transfer_rate=1000;
// This is set to 1Khz because of the fact that QNX default real-time clock is
running at 1 kHz.
/*********************************************************************/
/* Check all user arguments from the command-line */
for (i=1 ; i <argc;i++)
{
if (!strcmp(argv[i],"-v")) runmode=VERBOSE;
else if (!strcmp(argv[i],"-s")) runmode=SILENT;
else if (!strncmp(argv[i],"-b",2))
{
dev.iobase=strtoul((argv[i+1]),&check,16);
if (dev.iobase==0)
{
printf("PCM3718: Unkown base adress %s\n",argv[i+1]);
return 1;
}
i++;
}
else if (!strncmp(argv[i],"-d",2))
{
dev.dma_channel=strtoul((argv[i+1]),&check,10);
if (dev.dma_channel!=1 && dev.dma_channel!=3)
{
printf("PCM3718: Unknown DMA-channel %s\nAllowed DMA-channels are '1' or '3'\n",
argv[i+1]);
return EXIT_FAILURE;
}
i++;
}
else if (!strncmp(argv[i],"-c",2))
{
dev.CLK_speed=strtoul((argv[i+1]),&check,10);
if (dev.CLK_speed==1 || dev.CLK_speed==10)
{
dev.CLK_speed*=1000000;
}
else
{
printf("PCM3718: Unknown Clockspeed %s\nAllowed Clockspeeds are '1' or '10'
Mhz\n",argv[i+1]);
return EXIT_FAILURE;
}
i++;
}
else if (!strncmp(argv[i],"-i",2))
{
dev.IRQ=strtoul((argv[i+1]),&check,10);
if (dev.IRQ<2 || dev.IRQ>7)
{
printf("PCM3718: Unknown IRQ %s\nAllowed IRQs are '2' to '7'\n",argv[i+1]);
return EXIT_FAILURE;
}
i++;
}
else if (!strncmp(argv[i],"-r",2))
{
dev.transfer_rate=strtoul((argv[i+1]),&check,10);
if (dev.transfer_rate>100000||check-argv[i+1]==0)
{
printf("PCM3718: Unknown filterspeed %s\nMust be less than 100000\n",argv[i+1]);
return EXIT_FAILURE;
```

```
}
i++;
}
else
{
printf("unknown argument: %s\n\n",argv[i]);
printf("Usage: \npcm3718 [mode] [-b adress] [-d channel] [-c clockspeed] [-i irq]
[-r datarate]\n\n");
printf("runmodes:\n -n: normal mode \t(default)\n -v: verbose mode\n -s: silent
mode\n");
printf("configuration parameters:\n");
printf(" -b: IO-card baseadress, hexadecimal (default=210)\n");
printf(" -d: DMA-channel (default=3)\n");
printf(" -c: External clockspeed (default=10 Mhz)\n");
printf(" -i: Irq used (default=5)\n");
printf(" -r: Data transfer rate, Hertz (default=2Khz)\n");
return EXIT_FAILURE;
}
}

// Check that no pcm3718 device driver is already handling the /dev/pcm3718 device.
fd = open("/dev/PCM3718", O_RDWR | O_CREAT | O_EXCL);
if (errno==EEXIST)
{
printf("Error: /dev/PCM3718 device manager already exists!\nShutting down...\n");
return EXIT_FAILURE;
}
else if (errno !=ENOENT && runmode!=SILENT) {perror("PCM3718"); return
EXIT_FAILURE;}

if(PCM3718_detect(dev.iobase))
{
if (runmode!=SILENT) printf("Fatal Error:\nCard not detected at
%xh!\n",dev.iobase);
return EIO;
}

// Reset the state vector
status.device_open=0;
status.dma_set=0;
status.channel_set =0;
status.pacer_set =0;
status.ad_running=0;
status.filter_attached=0;


/* initialize dispatch interface */
if((dpp = dispatch_create()) == NULL && runmode!=SILENT) {
fprintf(stderr, "%s: Unable to allocate dispatch handle.\n",
argv[0]);
return EXIT_FAILURE;
}

/* initialize resource manager attributes */
memset(&resmgr_attr, 0, sizeof resmgr_attr);
resmgr_attr.nparts_max = 1;
resmgr_attr.msg_max_size = 2048;

/* initialize functions for handling messages */
iofunc_func_init(_RESMGR_CONNECT_NFUNCS, &connect_funcs,
_RESMGR_IO_NFUNCS, &io_funcs);

/* initialize attribute structure used by the device */
iofunc_attr_init(&attr, S_IFNAM | 0666, 0, 0);


/* attach our device name */
id = resmgr_attach(dpp, /* dispatch handle */
&resmgr_attr,              /* resource manager attrs */
"/dev/PCM3718",            /* device name */
_FTYPE_ANY,         /* open type */
```

G.7 pcm3718.c

```
0,                          /* flags */
&connect_funcs,             /* connect routines */
&io_funcs,                  /* I/O routines */
&attr);                     /* handle */
if(id == -1 && runmode!=SILENT) {
fprintf(stderr, "%s: Unable to attach name.\n", argv[0]);
return EXIT_FAILURE;
}

// Register all custom made function to handle device calls
io_funcs.close_ocb = io_close;
io_funcs.devctl = io_devctl; /* For handling _IO_DEVCTL, sent by devctl() */
io_funcs.write = io_write;
io_funcs.read = io_read;
connect_funcs.open = io_open;


if (runmode!=SILENT)
{
printf("PCM-3718 Device started...\n");
printf("Settings used:\n");
printf("Base adress: %xh ",dev.iobase);
printf("DMA-channel: %d ", dev.dma_channel);
printf("Internal Clockspeed: %dHz ",dev.CLK_speed);
printf("IRQ: %d \n",dev.IRQ);
printf("Transfer rate: %dHz\n",dev.transfer_rate);

}

/* allocate a context structure */
ctp = dispatch_context_alloc(dpp);
/* start the resource manager message loop */
while(1) {
if((ctp = dispatch_block(ctp)) == NULL && (runmode!=SILENT)) {
fprintf(stderr, "PCM3718: block error\n");
return EXIT_FAILURE;
}
dispatch_handler(ctp);
}
}
```

G.8 netserver.c

```
/**********TCP/IP Enabled Rocket Control Platform**************/
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <sys/netmgr.h>
#include <hw/inout.h>
#include<sys/mman.h>
#include<sys/neutrino.h>
#include <devctl.h>
#include <sys/iofunc.h>
#include "pcm3718.h"
#include "pulsecodes.h"
#include "IDList.h"
#include "ioports.h"
#include "servocontrol.h"

#define PORT 6322      /* the port users will be connecting to */
#define KEEPALIVEPORT 6323 /* Port for Keep alive data */
#define BACKLOG 10         /* how many pending connections queue will
                              hold */
#define MAXSENDSIZE 512    /* Size of TCP/IP send buffer */
#define MAXRECVLENGTH 512  /* Size of TCP/IP receive buffer */
#define MAXDATASIZE 256    /* Size of shared Command data memory area*/
#define DMABUFFERSIZE 100  /* Maximum size of the DMA-buffer used by
                              the filter */

const char HeaderStart[3]={0xFF,0xFE,0xFE};

int sockfd,servfd,card_fd,heartbeat_fd,servo;
/* listen on sockfd, new connection on servfd
PCM3718 Connection on card_fd
servo-controller on servo */

int channelid_send,channelid_filt;
int pulseprio; // A number used by all MsgSendPulse-commands

/********Shared Memory areas, Read-Write-Lock Protected **************/

/************** Analog data channels. ***************/
struct
{
double channel[16];
pthread_rwlock_t rwl;
} ad_data;
/***********************************************/

/***************** Command data *****************/
struct
{
unsigned short id;
char buffer[MAXDATASIZE];
pthread_rwlock_t rwl;
} Command_data;
/***********************************************/

/********************** Type defintions ****************************/
typedef union {
struct _pulse pulse;
/* your other message structures would go
here too */
} MSG;
```

G.8 netserver.c

```
/************************ Thread implementation ******************/

/*============================================
Keepalive Heartbeat thread.
This Thread sends small amounts of data on a separate socket to a
similar thread located in the network client program.
These messages makes sure both sides of the connection are "alive",
even when no "normal" data is sent over the transmission port.
*/
void * Heartbeat_thread(void* arg)
{
int nprobes;
int HBsockfd; /* listen on HBsock_fd, */
struct sockaddr_in my_addr; /* my address information */
struct sockaddr_in their_addr; /* connector's address information */
int sin_size,n,maxnalarms=5;
struct timeval timeout;
char c;
fd_set rfd;

if ((HBsockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
perror("Heartbeat:");
return;
}
my_addr.sin_family = AF_INET; /* host byte order */
my_addr.sin_port = htons(KEEPALIVEPORT); /* short, network byte order */
my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
bzero(&(my_addr.sin_zero), 8); /* zero the rest of the struct */
if (bind(HBsockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))== -1)
{
perror("HeartBeat");
close(HBsockfd);
return;
}

if (listen(HBsockfd, BACKLOG) == -1)
{
perror("Heartbeat");
close(HBsockfd);
return;
}
sin_size = sizeof(struct sockaddr_in);
if ((heartbeat_fd = accept(HBsockfd, (struct sockaddr *)&their_addr, &sin_size)) ==
-1)
{
perror("Hearbeat");
close(HBsockfd);
return;
}

printf("Server: got Keepalive Connection from %s\n",
inet_ntoa(their_addr.sin_addr));


FD_ZERO( &rfd );
FD_SET(heartbeat_fd, &rfd );
timeout.tv_sec=1;
timeout.tv_usec=0;
FD_ZERO( &rfd );

while(1)
{
FD_SET(heartbeat_fd, &rfd );

switch ( n = select( 1 +(heartbeat_fd),&rfd, 0, 0, &timeout ) )
{
case -1:
perror( "select" );
return;
case 0:
```

```
if (++nprobes > maxnalarms)
{
printf("Server: Client not responding!\n");
close(heartbeat_fd);
close(HBsockfd);
/* At this point the client is considered non-responding. What shall we do about
it??? */
return;
}
break;
default:
if ((n=recv(heartbeat_fd, &c,1,0)) <= 0 )
{
close(heartbeat_fd);
close(HBsockfd);
return;
}
nprobes=0;
send(heartbeat_fd,&c,1,0);
}

}
close(heartbeat_fd);
close(HBsockfd);
}



/*=====================================================
TCP/IP cleanup function
Note: These cleanup functions only gets called when calling ThreadCancel(). Or if
the thread shuts down itself. In other words, this code is not used. However, they
might become useful at a later stage.
*/
void Listen_Cleanup( void * arg )
{
close (servfd);
close(sockfd);
printf("Listen thread shutdown correctly\n");
usleep(2000);
}

/*=====================================================
TCP/IP listening thread.
Receives data from the client over TCP/IP network.
The thread first initializes the TCP/IP protocols and binds itself to a network
port, waiting
for incoming connections.
Then it simply blocks itself on a recv()-command, in an infinite loop.
When data has been received, the QNX OS automatically unblocks the
thread and it processes the data before blocking itself again.
*/

void * tcplistening_thread(void * arg)
{
struct sockaddr_in my_addr; /* my address information */
struct sockaddr_in their_addr; /* connector's address information */
int sin_size,value;
int numbytes,index,pos;
struct listen_timerevent;
struct itimerspec listen_itime;
char recvbuffer[MAXRECVLENGTH]; //Incoming tcp-ip packet buffer
char buffer[80];
char *ptr,*maxptr,*endptr;
ID_t PacketID;
int chid_filt, chid_send;

pthread_cleanup_push(&Listen_Cleanup, NULL );
if (pthread_rwlock_init(&Command_data.rwl,NULL)!=EOK)
fprintf(stderr,"Listen-thread: Error while initating rwlock\n");
```

G.8 netserver.c

```
// connect to filter thread
chid_filt=ConnectAttach(ND_LOCAL_NODE, 0,channelid_filt,_NTO_SIDE_CHANNEL, 0);
// connect to send thread
chid_send=ConnectAttach(ND_LOCAL_NODE, 0,channelid_send,_NTO_SIDE_CHANNEL, 0);


// Initialize the TCP/IP connection
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
{
perror("socket");
exit(1);
}
my_addr.sin_family = AF_INET; /* host byte order */
my_addr.sin_port = htons(PORT); /* short, network byte order */
my_addr.sin_addr.s_addr = INADDR_ANY; /* auto-fill with my IP */
bzero(&(my_addr.sin_zero), 8); /* zero the rest of the struct */

if (bind(sockfd, (struct sockaddr *)&my_addr, sizeof(struct sockaddr))== -1)
{
perror("bind");
exit(1);
}

if (listen(sockfd, BACKLOG) == -1)
{
perror("listen");
exit(1);
}
sin_size = sizeof(struct sockaddr_in);
// This loop just waits for connections from clients, if connection is closed it
returns here.
for(;;)
{
printf("Server: Waiting for client connection\n");
if ((servfd = accept(sockfd, (struct sockaddr *)&their_addr, &sin_size)) == -1)
{
perror("accept");
exit(0);
}
pthread_create (NULL, NULL,Heartbeat_thread, NULL);
printf("Server: got connection from %s\n",
inet_ntoa(their_addr.sin_addr));



/**************** TCP/IP Listening Loop ******************/
for(;;) // Listening for TCP data while connection is open
{
memset(buffer,0,sizeof(buffer));
if ((numbytes=recv(servfd, recvbuffer,sizeof(recvbuffer),0)) <=0 )
{
if (numbytes==0)
{
printf("Server: Connection closed by peer!\n");
close(servfd);
// The heartbeat server silently closes down when it's connection is broken
close(heartbeat_fd);
break;
}
else if (numbytes==-1)
{
perror("server");
close(servfd);
break;
}
}
else // numbytes >0 (i.e. we got some incoming data)
{
ptr=recvbuffer;
recvbuffer[numbytes] = '\0';
maxptr=ptr+numbytes;
```

```
printf("Received %d bytes of data\n",numbytes);

while (ptr<maxptr) // While there is still data left in buffer
{
index=0;

// Skip ahead to beginning of packet header

while (*ptr!=HeaderStart[index] && ptr<maxptr) ++ptr;
if (ptr==maxptr)
{
printf("Error: Data packet received, but no Packetheader was found");
continue;
}
// Check for correct packetheader
while (*ptr==HeaderStart[index] && ptr<maxptr)
{
++ptr;
if (++index==sizeof(HeaderStart)) break;
}
if (index!=sizeof(HeaderStart))
{
printf("Error: Data packet received, but no correct Packetheader …
was found");
continue;
}
if (ptr<(maxptr-2)) // We got Packet ID
{
memcpy(PacketID.asChar,ptr,2);
ptr+=2;
switch(ntohs(PacketID.asUintN))
{

/********* THIS IS THE INCOMING Packet/Type ID INTERPRETER **********/

case ID_RQIDNAMES: //Client is requesting IDs and names from Server.
printf("Request acknowledged\n");
MsgSendPulse (chid_send,pulseprio,COMMAND_PULSE_CODE,ID_RQIDNAMES);
break;

case ID_START_AD: // Client requests AD-conversion to begin
MsgSendPulse (chid_filt,pulseprio,COMMAND_PULSE_CODE,ID_START_AD);
MsgSendPulse (chid_send,pulseprio,COMMAND_PULSE_CODE,ID_START_AD);

// These lines are just an example how to send textmessages back
// to the client.
// First we must lock the resource we want to use.
pthread_rwlock_wrlock(&Command_data.rwl);
// Then we fill it with some data
Command_data.id=ID_TEXTMESSAGE;
sprintf(Command_data.buffer,"Starting A/D Conversion");
// Then we unlock the memory area so other threads can access it.
pthread_rwlock_unlock(&Command_data.rwl);
// Send a pulse to the send-thread to send some data away.
MsgSendPulse (chid_send,pulseprio,COMMAND_PULSE_CODE,ID_SENDDATA);
break;

case ID_STOP_AD: // Client requests AD-conversion to halt
MsgSendPulse (chid_filt,pulseprio,COMMAND_PULSE_CODE,ID_STOP_AD);
MsgSendPulse (chid_send,pulseprio,COMMAND_PULSE_CODE,ID_STOP_AD);
break;

case ID_SET_SERVO_0: // An example for remotely moving a servo.
pos=strtoul(ptr,&endptr,10);
if (pos>254 || endptr-ptr==0)
printf("Cannot set servo position: %s\n",ptr);
else
set_servo(servo,0,pos);
break;

case ID_OPEN_VALVE_0:
```

```
SetPPin(PIN2,1); // Since Ignition thread is controlling PIN1, let's use PIN2
break;

case ID_CLOSE_VALVE_0:
SetPPin(PIN2,0);
break;

/******** Add your new ID packet interpreters here **************/

// If no package handler could be found, we assume an error has
// occured.
default: printf("Unkown command receiced, ID=%d\n",ntohs(PacketID.asUintN));
} // Switch
} // If we got packet id
else
{
printf("Error: Data packet received, but no Packet ID was in it\n");
}

// Skip ahead to next packet
ptr+=strlen(ptr)+1;

} // while there is still data left in buffer
} // if numbytes > 0

} // for(;;)

// You will end up here if the client closes the connection. You should reset all
states.

// Stop ad-conversion if it is running
MsgSendPulse (chid_filt,pulseprio,COMMAND_PULSE_CODE,ID_STOP_AD);
} // while()
//All cleanup code goes here:
close(sockfd);
pthread_cleanup_pop( 1 );
return 0;
}
/*============================================================
A simple wrapper for adding analog data values to the send-buffer used in the
sending thread */

int addAnalogData2Buffer(char ** buffer,int ID,double dvalue)
{
int numbytes;
ID_t PacketID;

memcpy(*buffer,HeaderStart,sizeof(HeaderStart));
*buffer+=sizeof(HeaderStart); numbytes=sizeof(HeaderStart);
PacketID.asUintN=htons(ID);
memcpy(*buffer,PacketID.asChar,2);
*buffer+=2; numbytes+=2;
sprintf(*buffer,"%09.6f",dvalue);
numbytes+=strlen(*buffer)+1;
*buffer+=strlen(*buffer)+1;
return numbytes;
}



/*============================================================
TCP/IP sending thread.
Sends data to the client.
This thread is basically a huge message handler. It waits for messages from either
other
threads or timers. It then sends data to the client using tcp/ip.
The type of data depends on what message was received.
*/

void * tcpsending_thread(void * arg)
{
```

G.8 netserver.c

```
int r,value,rcvid,numbytes,nsend,i;
struct sigevent send_timerevent;
struct itimerspec send_itime;
timer_t send_timer_id;
MSG msg;
char sendbuffer[MAXSENDSIZE];
char *ptr;
ID_t PacketID;

// Create a comm. channel so other threads and timers can send message // to this
thread.
channelid_send = ChannelCreate(0);

// This is how you typically initalizes a timer in QNX
send_timerevent.sigev_notify = SIGEV_PULSE;
send_timerevent.sigev_coid =
ConnectAttach(ND_LOCAL_NODE, 0,channelid_send,_NTO_SIDE_CHANNEL, 0);
send_timerevent.sigev_priority = sched_get_priority_max(sched_getscheduler(0));
send_timerevent.sigev_code = TIMER_PULSE_CODE;
timer_create(CLOCK_REALTIME, &send_timerevent, &send_timer_id);
send_itime.it_value.tv_sec = 1;
send_itime.it_value.tv_nsec = 0;
send_itime.it_interval.tv_sec = 1;
send_itime.it_interval.tv_nsec = 0;

/*** The TCP Send thread is nothing but a message dispatcher ***/
for(;;)
{
rcvid = MsgReceive(channelid_send, &msg, sizeof(msg), NULL);
if (rcvid == 0)
{ /* we got a pulse */
if (msg.pulse.code == TIMER_PULSE_CODE)
// The timer in this case is used to
// send feedback data back to the client.
{
memset(sendbuffer,0,sizeof(sendbuffer));
ptr=sendbuffer;
nsend=0;
pthread_rwlock_rdlock(&ad_data.rwl);
// This example sends the values of analog channel 1 and two back to
// the client.
// You can simply add more channels just by adding lines like the ones // below.
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN0,ad_data.channel[0]);
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN1,ad_data.channel[1]);
nsend+=addAnalogData2Buffer(&ptr,ID_CHAN3,ad_data.channel[3]);
/************ Add more analog channel data here ******************/
pthread_rwlock_unlock(&ad_data.rwl);
if ((numbytes=send(servfd,sendbuffer,nsend,0)) <=0)
{
if (numbytes==-1)
{
printf("Server: Data feedback link aborted\n");
timer_delete(send_timer_id);
timer_create(CLOCK_REALTIME, &send_timerevent, &send_timer_id);
}
if (numbytes==0) {printf("sending error (0 bytes sent)\n");}
}
// printf("Sent %d bytes of measurement data\n",nsend);
} // End of TIMER_PULSE_CODE

else if (msg.pulse.code == COMMAND_PULSE_CODE)
{
// a COMMAND_PULSE_CODE means that some other thread is relaying a
// command to the threads of interest.
// Instead of using some internal message-type-structure.
// The same structure is used by the tcp-communication.
// In most cases, the listen-thread is the one sending these commands.
// It does this to notify all other threads that a certain event has
// occured.
// How each thread interprets what to do when these events/commands
// occur is totally indivdual.
```

```
switch(msg.pulse.value.sival_int)
{
case ID_START_AD:
// Start the feedback timer (the one generating TIMER_PULSE_CODE pulses)
timer_settime(send_timer_id, 0, &send_itime, NULL);
break;
case ID_STOP_AD: // Resets the feedback timer
timer_delete(send_timer_id);
timer_create(CLOCK_REALTIME, &send_timerevent, &send_timer_id);
break;
case ID_RQIDNAMES:
// This means that the client has requested the names and id number of
// all members in the IDMAP (see IDList.h).
memset(sendbuffer,0,sizeof(sendbuffer));
// printf("Ready to transmit IDs and Names\n");
for(i=0;i<IDMAP_SIZE ;i++)
{
PacketID.asUintN=htons(ID_TXIDNAME);
ptr=sendbuffer;
strcpy(ptr,HeaderStart);
ptr+=sizeof(HeaderStart);
memcpy(ptr,PacketID.asChar,2);
ptr+=2;
sprintf(ptr,"%s %d",IDMAP[i].ID_Name,IDMAP[i].ID_Value);
ptr+=strlen(ptr);
nsend=(int) (ptr-sendbuffer+1);
numbytes=send(servfd,sendbuffer,nsend,0);
printf("sent IDName with %d bytes of data\n",nsend);
}
PacketID.asUintN=htons(ID_IDNAMECOMPLETE);
ptr=sendbuffer;
strcpy(ptr,HeaderStart);
ptr+=sizeof(HeaderStart);
memcpy(ptr,PacketID.asChar,2);
ptr+=2;
*ptr='\0';
nsend=(int) (ptr-sendbuffer+1);
numbytes=send(servfd,sendbuffer,nsend,0);
// printf("sent IDNameComplete with %d bytes of data\n",nsend);
break;
// All the commands above didn't use any parameters except for the fact
// that a certain event has occured.
// In order to pass more than one parameters to COMMAND_PULSE_CODE
// commands you must use the Command_data shared memory resource.
// It should be the listen_threads job to package the data into this
// memory area.

case ID_SENDDATA: // This command sends the transmission package placed
// in the Command_datameory area.
pthread_rwlock_rdlock(&Command_data.rwl);
PacketID.asUintN=htons(Command_data.id);
ptr=sendbuffer;
strcpy(ptr,HeaderStart);
ptr+=sizeof(HeaderStart);
memcpy(ptr,PacketID.asChar,2);
ptr+=2;
strcpy(ptr,Command_data.buffer);
pthread_rwlock_unlock(&Command_data.rwl);
ptr+=strlen(ptr);
nsend=(int) (ptr-sendbuffer+1);
numbytes=send(servfd,sendbuffer,nsend,0);
break;
}
}
}
// else other messages
} // for(;;)
}
/*============================================================
Mutex destroy function
```

G.8 netserver.c

```
Called by the filter thread during cleanup
*/
void RW_Destroy( void * arg )
{
usleep(2000);
pthread_rwlock_unlock(&ad_data.rwl);
if(pthread_rwlock_destroy(&ad_data.rwl)==EBUSY) printf("Error: Write lock was found
in a busy state\n");
}

/*=====================================================================
The Analog filter thread
This is a simple digital measuremnt noise filter.

Only this thread is in control of the data acquisition card (PCM3718).
Other threads can in a way control the card by sending messages to this thread. But
it is up
to this thread how, when and if these messages will be executed.
*/

void * filter_thread (void *arg)
{
unsigned char buffer[DMABUFFERSIZE];
unsigned char *dmabuf;
channel_setup_data_t channel_setup_data;
typedef enum {BIPOLAR,UNIPOLAR} polartype_t;
// These two arrays are used for the calculation of the voltage number of a
channel.
const polartype_t polartype[9] = {BIPOLAR,BIPOLAR, BIPOLAR,BIPOLAR, UNIPOLAR,
UNIPOLAR,UNIPOLAR,UNIPOLAR,BIPOLAR};
const double rangevalues[9]={10.0,5.0,2.5,1.25,10.0,5.0,2.5,1.25,20.0};
int AD_is_running=0; // An internal state variable to keep track of the
                     // A/D converter.
int ret,rcvid,i,fd2,N,channelnr,intval;
unsigned int bufsize;
attach_message_t fm;
MSG msg;
unsigned char *ptr,*bufferend;
double alpha1,alpha2,dval;
double_uint pacerdata;
unsigned int filter_speed=2000;
// This filter is filtering 2000 times / Second

// The thread starts with initializing the Data acquisition card
// The outline here should be followed for all programs that uses the
// QNX PCM3718 driver.

// Clear the memory
memset(&channel_setup_data,0,sizeof(channel_setup_data));
// Create a channel so other threads can send messages to this one.
channelid_filt = ChannelCreate(0);
// Sets the speed to 100Khz (assuming a 10 MHz external clock)
pacerdata.c1=10; pacerdata.c2=10;
channel_setup_data.start=0; // Start channel for conversion is 0
channel_setup_data.stop=4; // Stop channel for conversion is 4
// (giving a total of 5 channels)
for (i=0;i<15;i++)
// Sets all first 16 channels to the same conversion range.
// Note that only channel 0-4 wil be set by the driver because
// it uses the start/stop parameters as boundaries.
{
channel_setup_data.range[i]=UNIPOLAR_10V;
}

// Open the device
if ((card_fd = open("/dev/PCM3718", O_RDWR)) == -1)
{
printf("Server: unable to open device manager\n");
exit(1);
}
```

G.8 netserver.c

```
// Prepare info for the driver to wake this thread up
fm.data.channelid=channelid_filt;
fm.data.pidid=getpid();


// Now set up the card as desired.
// Pacer speed (100 Khz)
ret = devctl(card_fd, DEVCTL_SETPACER, &pacerdata, sizeof(pacerdata), NULL);
// Channel ranges
ret = devctl(card_fd, DEVCTL_SETCHANNEL, &channel_setup_data,
sizeof(channel_setup_data), NULL);
// Transfer speed (2 KHz)
ret = devctl(card_fd, DEVCTL_SET_TRANSFER_SPEED, &filter_speed,
sizeof(filter_speed), NULL);
// Initialize the DMA-controller
ret = devctl(card_fd, DEVCTL_DMA_INIT, NULL, 0, NULL);
// Attach this thread to the driver for synchronization
ret = devctl(card_fd, DEVCTL_ATTACHFILTER, &fm, sizeof(fm), NULL);

// The filter is a simple first order low-pass filter.
// Also known as exponentially moving average filter.
N=7; // Average over 7 samples (some arbitrary number)
alpha1=1.0 / (N+1.0); // Filter parameter
alpha2=(double) N / (N+1.0); // Filter parameter

if(pthread_rwlock_init(&ad_data.rwl,NULL)!=EOK)
fprintf(stderr,"Filter thread: Error while initating rwlock\n");
pthread_cleanup_push( &RW_Destroy, NULL );

// Clear the analog channel data-area.
pthread_rwlock_wrlock(&ad_data.rwl);
memset(ad_data.channel,0,sizeof(ad_data.channel));
pthread_rwlock_unlock(&ad_data.rwl);

// Prepare the buffer pointers
bufsize=fm.reply.size;
bufferend=buffer+bufsize;

// Open the shared memory area.
if ((fd2=shm_open(fm.reply.shm_name, O_RDWR, 0770)) == -1)
{
fprintf(stderr,"CLIENT: ERROR cannot create shmem area");
close(fd2);
abort();
}


else if( (dmabuf = mmap(NULL,bufsize,
PROT_READ | PROT_WRITE | PROT_NOCACHE,
MAP_SHARED, fd2, 0)
) == MAP_FAILED)
{
perror("mmap");
close(fd2);
abort();
}
close(fd2);
// we are now ready to start filtering data.
for (;;)
{
rcvid = MsgReceive(channelid_filt, &msg, sizeof(msg), NULL);
if (rcvid == 0)
{
if (msg.pulse.code == FILTER_PULSE_CODE)
// The driver has notified us that it is now safe to go get some data.
{
memcpy(buffer,dmabuf,bufsize); // get the raw data
pthread_rwlock_wrlock(&ad_data.rwl);

// Convert the raw data into actual voltages.
for(ptr=buffer;ptr<bufferend;ptr+=2)
```

```
{
channelnr=*ptr&0x0f;
intval=((unsigned short) *(ptr+1))<<4|(*(ptr)>>4);
dval=intval/4096.0*rangevalues[channel_setup_data.range[channelnr]];
if (polartype[channel_setup_data.range[channelnr]]==BIPOLAR)
dval-=rangevalues[channel_setup_data.range[channelnr]]/2;
ad_data.channel[channelnr]=alpha1*dval+alpha2*ad_data.channel[channelnr];
}
pthread_rwlock_unlock(&ad_data.rwl);
}
// Else some other thread is trying to send some message/event/command
else if (msg.pulse.code == COMMAND_PULSE_CODE)
{
switch(msg.pulse.value.sival_int)
{
case ID_START_AD:
if (!AD_is_running)
{
devctl(card_fd, DEVCTL_AD_START, NULL, 0, NULL);
AD_is_running=1;
}
break;
case ID_STOP_AD:
if (AD_is_running)
{
devctl(card_fd, DEVCTL_AD_STOP, NULL, 0, NULL);
AD_is_running=0;
}
break;
}
}
else if (msg.pulse.code == SHUTDOWN_PULSE_CODE) break;
}
//else other messages
} // for loop
if (munmap(dmabuf,bufsize)==-1)
printf("CLIENT: Error releasing shared memory!\n");
pthread_cleanup_pop( 1 );
}
/*=====================================================================
Ignition Thread
Once the hardware is ready, this thread should contain a synchronized script
controlling the thrust of the rocket motor.
Meanwhile, this thread generates a squarewave-pulse on the parallel-port, wich
switches state every 2 seconds. This is to demonstrate the use of timers and I/O
control.
*/
void * ignition_thread (void *arg)
{
struct sigevent timerevent;
struct itimerspec itime;
timer_t timer_id;
struct _clockperiod clockp,oldclock;
int state=0;
int channelid_ignition,rcvid;
MSG msg;

channelid_ignition = ChannelCreate(0);
timerevent.sigev_notify = SIGEV_PULSE;
timerevent.sigev_coid = ConnectAttach(ND_LOCAL_NODE,
0,channelid_ignition,_NTO_SIDE_CHANNEL, 0);
timerevent.sigev_priority = sched_get_priority_max(sched_getscheduler(0));
timerevent.sigev_code = TIMER_PULSE_CODE;
timer_create(CLOCK_REALTIME, &timerevent, &timer_id);
itime.it_value.tv_sec = itime.it_interval.tv_sec = 2;
itime.it_value.tv_nsec = itime.it_interval.tv_nsec = 0 ;
timer_settime(timer_id, 0, &itime, NULL);

for(;;)
{
rcvid = MsgReceive(channelid_ignition, &msg, sizeof(msg), NULL);
```

```
if (rcvid == 0)
{ /* we got a pulse */
if (msg.pulse.code == TIMER_PULSE_CODE)
{
if(state==0)
{
SetPPin(PIN1,1);    // Example of Parallell-port control
SetPinHigh(PIN1,1); // Example of DAQ IO-port control
state=1;
}
else
{
SetPPin(PIN1,0);
SetPinHigh(PIN1,0);
state=0;
}
}
}
}
}

/*=====================================================================
Control thread */
void * Control_thread (void *arg)
{

}

/*=====================================================================
File logging thread
This thread demonstrates how a file-logging tread can be constructed.
*/

void * Logging_thread (void *arg)
{
struct sigevent timerevent;
struct itimerspec itime;
timer_t timer_id;
struct _clockperiod clockp,oldclock;
int i;
int channelid_ignition,rcvid;
MSG msg;
char buffer[80];
FILE *fp;
char *ptr;
fp=fopen("./logdata","w");

channelid_ignition = ChannelCreate(0);
timerevent.sigev_notify = SIGEV_PULSE;
timerevent.sigev_coid = ConnectAttach(ND_LOCAL_NODE,
0,channelid_ignition,_NTO_SIDE_CHANNEL, 0);
timerevent.sigev_priority = sched_get_priority_max(sched_getscheduler(0));
timerevent.sigev_code = TIMER_PULSE_CODE;
timer_create(CLOCK_REALTIME, &timerevent, &timer_id);
itime.it_value.tv_sec = itime.it_interval.tv_sec = 1;
itime.it_value.tv_nsec = itime.it_interval.tv_nsec = 0;
timer_settime(timer_id, 0, &itime, NULL);

for(;;)
{
rcvid = MsgReceive(channelid_ignition, &msg, sizeof(msg), NULL);
if (rcvid == 0)
{ /* we got a pulse */
if (msg.pulse.code == TIMER_PULSE_CODE)
{
pthread_rwlock_rdlock(&ad_data.rwl);
ptr=buffer;
for (i=0;i<2;i++)
{
sprintf(ptr,"Channel %d: %9.6f\r\n",i,ad_data.channel[i]);
ptr+=strlen(ptr);
```

G.8 netserver.c

```
}
sprintf(ptr,"\r\n");
pthread_rwlock_unlock(&ad_data.rwl);
fprintf(fp,buffer);
fflush(fp);
}
}
}
fclose(fp);
}

/*=========================================================================
netserver main function
*/
main()
{
int policy;
struct sched_param param;
pthread_t listen_thread_id,send_thread_id,filter_thread_id;
pthread_attr_t attr;
ThreadCtl(_NTO_TCTL_IO, 0); //Enable this process (and it's threads) to
                            //access the computer hardware.
                           // This is not required to access the AD-
                          // Card since it is taken care of using the
                         // pcm3718 device manager. However, this is
                        // needed if you are controlling I/O-ports.

//Initialize the servo-controller on Com-port 1
if((servo=servo_open("/dev/ser1"))<0) perror("servo_open");


pulseprio=sched_get_priority_max(sched_getscheduler(0))-1;


// Select priorities and initialize threads.
pthread_attr_init(&attr);
pthread_getschedparam(pthread_self(),&policy,&param);
// pthread_getschedparam(filter_thread_id,&policy,&param);
// printf("Priority is %d\n",param.sched_priority);
pthread_attr_setinheritsched(&attr,PTHREAD_EXPLICIT_SCHED);

/* Create the filter thread */
param.sched_priority=12;
pthread_attr_setschedparam(&attr,&param );
pthread_create (&filter_thread_id, &attr, filter_thread,NULL); usleep(5000);

/* Create the Control thread */
param.sched_priority=15;
pthread_attr_setschedparam(&attr,&param );
pthread_create (NULL, &attr,Control_thread, NULL); usleep(5000);

/* Create other non-default-priority-threads here using the 3 lines of code above
as
a template */

/* Create the other threads */
pthread_create (NULL, NULL,ignition_thread, NULL); usleep(5000);
pthread_create (NULL, NULL,tcpsending_thread, NULL); usleep(5000);
pthread_create (NULL, NULL,Logging_thread, NULL); usleep(5000);
pthread_create (&listen_thread_id, NULL,tcplistening_thread, NULL);

// Wait until Listen thread terminates (=Wait forever);
pthread_join(listen_thread_id,NULL);
}
```

G.9 netclient.c

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <netinet/in.h>
#include <sys/socket.h>
#include <netdb.h>
#include <sys/wait.h>
#include <pthread.h>
#include <sys/mman.h>
#include <inttypes.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sched.h>
#include <time.h>
#include <unistd.h>
#include <stddef.h>
#include "IDList.h"

#define PORT 6322 /* the port client will be connecting to */
#define KEEPALIVEPORT 6323 /* Port for Keep alive data */

#define MAXDATASIZE 100 /* max number of bytes we can get at once */
#define MAXSENDSIZE 200

static int clifd,nsec,maxnprobes,nprobes,channelid;

const char HeaderStart[3]={0xFF,0xFE,0xFE};

void * Heartbeat_thread(void* arg)
{

int sockfd; /* connect on sock_fd*/
struct hostent *he;
struct sockaddr_in their_addr; /* server's address information */
struct timeval timeout;
fd_set rfd;
int n,value;
char c;

he=(struct hostent *) arg;

if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
perror("Heartbeat");
return;
}


their_addr.sin_family = AF_INET; /* host byte order */
their_addr.sin_port = htons(KEEPALIVEPORT); /* short, network byte
                                               order */
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
bzero(&(their_addr.sin_zero), 8); /* zero the rest of the struct */

if (connect(sockfd, (struct sockaddr *)&their_addr, \
sizeof(struct sockaddr)) == -1)
{
perror("Heartbeat");
close(sockfd);
return;
}


timeout.tv_sec=0;
timeout.tv_usec=0;
```

G.8 netclient.c

```
FD_ZERO( &rfd );

while(1)
{
FD_SET(sockfd, &rfd );
n=send(sockfd,"1",1,0);
usleep(1000000);
switch ( n = select( 1 +(sockfd),&rfd, 0, 0, &timeout ) )
{
case -1:
perror( "select" );
return;
case 0:
if (++nprobes > maxnprobes)
{
printf("Client: Server not responding!\n");
close(sockfd);
return;
}
break;
default:
if ((n=recv(sockfd, &c,1,0)) <= 0 )
{
close(sockfd);
return;
}
nprobes=0;
}

}
close(sockfd);
}

void * tcplistening_thread(void * arg)
{
int numbytes,index,servo,pos;
struct listen_timerevent;
struct itimerspec listen_itime;
char recvbuffer[200]; //Incoming tcp-ip packet buffer
char buffer[200];
char *ptr,*maxptr,*endptr;
ID_t PacketID;
int chid_filt, chid_send;

for(;;)
{
memset(buffer,0,sizeof(buffer));
if ((numbytes=recv(clifd, recvbuffer,80,0)) <=0 )
{
if (numbytes==0) {printf("Connection lost!\n"); close(clifd); break;}
else if (numbytes==-1) {perror("server"); break;}
}
else // numbytes >0
{
ptr=recvbuffer;
recvbuffer[numbytes] = '\0';
maxptr=ptr+numbytes;
printf("Received %d bytes of data\n",numbytes);

while (ptr<maxptr) // While there is still data left in buffer
{
index=0;
// Skip ahead to beginning of packet header

while (*ptr!=HeaderStart[index] && ptr<maxptr) ++ptr;
if (ptr==maxptr)
{
printf("Error: Data packet received, but no Packetheader was found");
continue;
}
// Check for correct packetheader
```

G.8 netclient.c

```
while (*ptr==HeaderStart[index] && ptr<maxptr)
{
++ptr;
if (++index==sizeof(HeaderStart)) break;
}
if (index!=sizeof(HeaderStart))
{
printf("Error: Data packet received, but no correct Packetheader was found");
continue;
}
if (ptr<(maxptr-2)) // We got Packet ID
{
memcpy(PacketID.asChar,ptr,2);
ptr+=2;
switch(ntohs(PacketID.asUintN))
{

/***** THIS IS THE INCOMING PACKET ID INTERPRETER ******/
/******** Add your new ID packet interpreters here ************/

default: printf("ID=%d\tValue=%s\n",ntohs(PacketID.asUintN),ptr);
} // Switch
} // If we got packet id
else
{
printf("Error: Data packet received, but no Packet ID was in it\n");
}

// Skip ahead to next packet
ptr+=strlen(ptr)+1;

} // while there is still data left in buffer
} // if numbytes > 0
} // for(;;)

//All cleanup code goes here:\
return 0;
}

int BuildCommandPacket(char *buffer,int ID,const char* data)
{
int numbytes;
ID_t IDnumber;
char *ptr=buffer;
memset(buffer,0,sizeof(buffer));
memcpy(ptr,HeaderStart,sizeof(HeaderStart));
ptr=buffer+sizeof(HeaderStart); numbytes=sizeof(HeaderStart);
IDnumber.asUintN=htons(ID);
memcpy(ptr,IDnumber.asChar,2);
ptr+=2; numbytes+=2;
strcpy(ptr,data);
numbytes+=strlen(data)+1;
return numbytes;
}

int main(int argc, char *argv[])
{
int sockfd,value,chid,numbytes;
struct hostent *he;
struct sockaddr_in their_addr; /* server's address information */
char sendbuffer[MAXSENDSIZE+1];
pthread_t listen_thread_id,heartbeat_thread_id;

if (argc != 2) {
fprintf(stderr,"usage: client hostname\n");
exit(1);
}
if ((he=gethostbyname(argv[1])) == NULL) { /* get the host info */
herror("gethostbyname");
exit(1);
}
```

G.8 netclient.c

```
if ((sockfd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
perror("socket");
exit(1);
}

their_addr.sin_family = AF_INET; /* host byte order */
their_addr.sin_port = htons(PORT); /* short, network byte order */
their_addr.sin_addr = *((struct in_addr *)he->h_addr);
bzero(&(their_addr.sin_zero), 8); /* zero the rest of the struct */

if (connect(sockfd, (struct sockaddr *)&their_addr, \
sizeof(struct sockaddr)) == -1) {
perror("connect");
exit(1);
}
clifd=sockfd;

// Send Command to server
value=BuildCommandPacket(sendbuffer,ID_START_AD,"");
if ((numbytes=send(clifd,sendbuffer,value,0)) <=0)
{
if (numbytes==-1) {perror("server");}
if (numbytes==0) {printf("sending error (0 bytes sent)\n");}
}


pthread_create (&listen_thread_id, NULL,tcplistening_thread, NULL);
usleep(5000);
pthread_create (&heartbeat_thread_id, NULL,Heartbeat_thread, he);
value=pthread_join(listen_thread_id,NULL);
printf("Closed\n");
close(sockfd);
return 0;
}
```

H Matlab Code

H.1 findGuidata.m

```
function [GUIhandles,GUIObject]=findGuidata(Tag)
%Finds the guidata handles for a running GUI application
%
%'Tag' is a string containing a UNIQUE name
%of a known fieldname inside your GUI handle structure.
%
%Example:
%Assume that you have a handlefield in your application
%called handles.Myhandlefield.
%Simply write:
%handles=findGuidata('Myhandlefield') or
%[GUIhandles,GUIObject]=findGuidata('Myhandlefield')
%if you require the object handle as well.
%
%Remarks:
%It is important that your tagname is unique.
%Don't search for 'pushbutton1' or 'edit1' or
%Such names that is common for GUI applications.
%If you do, you might end up retrieving the handles for the wrong
%application!
%
%findGuidata(Tag) returns 0 if the specific handle cannot be found.
function [GUIhandles,GUIObject]=findGuidata(Tag)
GUIhandles=0;
GUIObject=0;
if (ischar(Tag))
    h=allchild(0);  index=1;
    if (length(h)<1) return; end;
    while(index<=length(h) &&
isfield(getappdata(h(index),'UsedByGUIData_m'),Tag)==0)
        index=index+1;
    end
    if (index>length(h))
        return;
    else
        GUIhandles=guidata(h(index));
        GUIObject=h(index);
    end
else fprintf('Error in findGuidata: Tag must be a string!');
end
```

H.2 KeepaliveHandler.m

```
function KeepAliveHandler (varargin)
%Find the rocketclient handles using the unique tag 'COMh'
[apphandles,hObject]=findGuidata('COMh');
invoke(apphandles.COMh,'KeepAlive');
```

H.3 ConnectionstatusHandler.m

```
function ConnectionstatusHandler (varargin)
%Find the rocketclient handles using the unique tag 'COMh'
[apphandles,hObject]=findGuidata('COMh');
status=varargin{3};
if (status==1)
%Connected, disable connect button and enable disconnect button
    set(apphandles.pushbutton1,'Enable','off');
    set(apphandles.pushbutton2,'Enable','on');
    apphandles.connected=1;
else
    set(apphandles.pushbutton1,'Enable','on');
    set(apphandles.pushbutton2,'Enable','off');
    apphandles.connected=0;
end
guidata(hObject,apphandles);
```

H.4 DatarecvdHandler.m

```
% Event Handler function from RocketNetworkProxy
function DatarecvdHandler (varargin)
%Find the rocketclient handles using the unique tag 'COMh'
[apphandles,hObject]=findGuidata('COMh');
IDs=varargin{3};
Values=varargin{4};

%Add value to workspace
for i=1:length(IDs);
    IDName=lower(IDs{i}); %The name is now in lowercase
    if(IDName(1:3)=='id_') IDName=IDName(4:length(IDName)); end;
    if (~evalin('base',['exist(''' IDName ''');']))
        evalin('base',[IDName '=[];']);
    end
    %If the data value is a numerical value
    if(length(str2num(Values{i})>0))
        evalin('base',[IDName '=[' IDName ';' Values{i} '];']);
    else % The data value is non-numeric, store it in a cellarray
    evalin('base',[IDName '{1,length(' IDName  ')+1}=''' Values{i}''';']);
    end

%Add value to GUI
    if (isfield(apphandles,IDs{i}))
        if(length(str2num(Values{i})>0))
        eval(['set(apphandles.' IDs{i} ',''String'',' Values{i}');']);
        else
        eval(['set(apphandles.' IDs{i} ',''String''',''' Values{i}''');']);
        end
        guidata(hObject,apphandles);
    end
end
```

H.5 rocketclient.m

```
function varargout = rocketclient(varargin)
% ROCKETCLIENT M-file for rocketclient.fig
% ROCKETCLIENT, by itself, creates a new ROCKETCLIENT or raises the
% existing singleton*.
%
% H = ROCKETCLIENT returns the handle to a new ROCKETCLIENT or the
% handle to the existing singleton*.
%
% ROCKETCLIENT('CALLBACK',hObject,eventData,handles,...) calls the
% local function named CALLBACK in ROCKETCLIENT.M with the given input %
arguments.
%
% ROCKETCLIENT('Property','Value',...) creates a new ROCKETCLIENT or
% raises the existing singleton*. Starting from the left, property
% value pairs are applied to the GUI before
% rocketclient_OpeningFunction gets called.  An
% unrecognized property name or invalid value makes property
% application stop.
% All inputs are passed to rocketclient_OpeningFcn via varargin.
%
% *See GUI Options on GUIDE's Tools menu.  Choose "GUI allows only one
% instance to run (singleton)".
%
% Begin initialization code - DO NOT EDIT
gui_Singleton = 1;
gui_State = struct('gui_Name',       mfilename, ...
                   'gui_Singleton',  gui_Singleton, ...
                   'gui_OpeningFcn', @rocketclient_OpeningFcn, ...
                   'gui_OutputFcn',  @rocketclient_OutputFcn, ...
                   'gui_LayoutFcn',  [] , ...
                   'gui_Callback',   []);
if nargin & isstr(varargin{1})
    gui_State.gui_Callback = str2func(varargin{1});
end

if nargout
    [varargout{1:nargout}] = gui_mainfcn(gui_State, varargin{:});
else
    gui_mainfcn(gui_State, varargin{:});
end
% End initialization code - DO NOT EDIT


% --- Executes just before rocketclient is made visible.
function rocketclient_OpeningFcn(hObject, eventdata, handles, varargin)
% This function has no output args, see OutputFcn.
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
% varargin   command line arguments to rocketclient (see VARARGIN)

% Choose default command line output for rocketclient
handles.output = hObject;

set(hObject,'CloseRequestFcn',{@rocketclient_ClosingFcn,handles})
% The lines below creates the ActiveX-control 'RocketNetworkProxy'. The
% handle COMh is used as a calling handle when issuing commands to the
% control. The size of the window is 350 by 198 pixels starting 40
% pixels up and 40 pixels to the right from the lower left corner of
% the client window.
```

H.5 rocketclient.m

```
% The registerevent-command registers the eventhandlers used by the
% client.
handles.COMh =
actxcontrol('RocketNetworkproxy.InDataControl', [40 40 350 198]);
registerevent(handles.COMh,{'datarecvd','DatarecvdHandler'; ...
'connectionstatus','ConnectionstatusHandler'; ...
'ping','KeepAliveHandler'});

handles.connected=0;

% Update handles structure
guidata(hObject, handles);


function rocketclient_ClosingFcn(hObject, eventdata, handles)
if(isfield(handles,'COMh'))
    if (handles.connected)
        invoke(handles.COMh,'Disconnect');
    end
    unregisterallevents(handles.COMh);
    delete(handles.COMh);
end;
delete(hObject);
evalin('base','clear apphandles');


% --- Outputs from this function are returned to the command line.
function varargout = rocketclient_OutputFcn(hObject, eventdata, handles)
% varargout  cell array for returning output args (see VARARGOUT);
% hObject    handle to figure
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Get default command line output from handles structure
varargout{1} = handles.output;

% --- Executes on button press in pushbutton1.
function pushbutton1_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
invoke(handles.COMh,'Connect','169.254.13.200');

% --- Executes on button press in pushbutton2.
function pushbutton2_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
%            invoke(handles.COMh,'Disconnect');


% --- Executes during object creation, after setting all properties.
function ID_CHAN0_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ID_CHAN0 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
%            called

% Hint: edit controls usually have a white background on Windows.
%       See ISPC and COMPUTER.
if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
```

```
end



function ID_CHAN0_Callback(hObject, eventdata, handles)
% hObject    handle to ID_CHAN0 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% Hints: get(hObject,'String') returns contents of ID_CHAN0 as text
%        str2double(get(hObject,'String')) returns contents of ID_CHAN0
%        as a double


% --- Executes on button press in pushbutton3.
function pushbutton3_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
if (handles.connected)
    invoke(handles.COMh,'SendCommand','ID_START_AD','');
    guidata(hObject, handles);
end


% --- Executes during object creation, after setting all properties.
function ID_CHAN2_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ID_CHAN2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
%            called

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end


function ID_CHAN2_Callback(hObject, eventdata, handles)
% hObject    handle to ID_CHAN2 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% --- Executes during object creation, after setting all properties.
function edit3_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
% called

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function edit3_Callback(hObject, eventdata, handles)
% hObject    handle to edit3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
```

```
% --- Executes during object creation, after setting all properties.
function ID_CHAN1_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ID_CHAN1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns called

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end



function ID_CHAN1_Callback(hObject, eventdata, handles)
% hObject    handle to ID_CHAN1 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% --- Executes during object creation, after setting all properties.
function edit4_CreateFcn(hObject, eventdata, handles)
% hObject    handle to edit4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
%            called

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end
function edit4_Callback(hObject, eventdata, handles)
% hObject    handle to edit4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% --- Executes on button press in pushbutton4.
function pushbutton4_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton4 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

invoke(handles.COMh,'SendCommand','ID_SET_SERVO_0',get(handles.edit4...
,'String'));

% --- Executes during object creation, after setting all properties.
function ID_CHAN3_CreateFcn(hObject, eventdata, handles)
% hObject    handle to ID_CHAN3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    empty - handles not created until after all CreateFcns
%            called

if ispc
    set(hObject,'BackgroundColor','white');
else
    set(hObject,'BackgroundColor',get(0,'defaultUicontrolBackgroundColor'));
end

function ID_CHAN3_Callback(hObject, eventdata, handles)
% hObject    handle to ID_CHAN3 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)

% --- Executes on button press in pushbutton8.
```

H.5 rocketclient.m

```
function pushbutton8_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton8 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
invoke(handles.COMh,'SendCommand','ID_OPEN_VALVE_0','');


% --- Executes on button press in pushbutton9.
function pushbutton9_Callback(hObject, eventdata, handles)
% hObject    handle to pushbutton9 (see GCBO)
% eventdata  reserved - to be defined in a future version of MATLAB
% handles    structure with handles and user data (see GUIDATA)
invoke(handles.COMh,'SendCommand','ID_CLOSE_VALVE_0','');
```